WordPress Deep Dive

Author: Wiktor Jarosz
First published date: November 1, 2025
Last modified date: November 1, 2025

This document is not a tutorial on creating WordPress websites. As a matter of fact, you'll still probably not be able to get a WordPress website live after reading it. Instead, it's a deep dive into the technical architecture of WordPress.

You'll learn how WordPress works below the hood. The goal of this document is to give you a deep understanding of the entire system. No matter if you're a plugin developer, a theme author, a freelancer, or want to contribute to the core - this guide will be the most significant WordPress resource you will ever read - and that's a promise.

To get the most out of it, I recommend that you play with the concepts described here on a local WordPress install while reading. This will make it more likely that you actually remember these concepts.

The guide is aimed at late beginner/intermediate+ web developers. I'm assuming a certain level of understanding of web servers, programming languages, websites, and other technical concepts. I usually don't explain things that are outside the scope of a "WordPress Deep Dive". If you find your knowledge gap prevents you from fully grasping a concept - go fill that gap first.

No prior knowledge of WordPress is required, but I suspect that you'll be better off if you spend just 5 minutes adding posts and jumping around the WordPress admin panel (or just watch a youtube video). It might help you ground your understanding in something you've seen with your own eyes. Having some context, even if minimal, is important for your brain not to have to wander around imagining what the thing looks like.

The guide is structured sequentially - that is, I'm assuming you've read chapters 1 and 2 before you read chapter 3. I sometimes refer to previous chapters when explaining new topics. You may try reading the chapters out of order, but to get the most value out of this document, I recommend you follow it from start to finish.

The code snippets in this document are not supposed to be blindly copy-pasted. Most of them are not robust, safe, or following best practices. Some of them may outright not work. They are included for demonstration purposes only, and that requires cutting out all the fluff that might interfere with the topic I'm trying to demonstrate.

This document is not supposed to be a replacement for the official documentation. It is just specific enough for you to understand the topics, but not so much that they become overwhelming. If you are in active need of the most up-to-date information on a given concept - always consult the documentation.

If you're completely new to WordPress - don't try to read this in a day. This document has 280 pages of complex technical content. Space it out evenly over some period of time. Otherwise, you won't remember anything from it.

With that introduction behind us, I welcome you to what I hope is the beginning of a long and exciting journey of discovering the beauty of WordPress.

What Is WordPress?

WordPress is a content management system (CMS) based traditionally on the LAMP stack (Linux, Apache, MySQL, PHP). Nginx can, and often is, used instead of Apache - WordPress is web server agnostic. Similarly, MariaDB can be used instead of MySQL.

Everything in WordPress is a post, kind of like everything in Unix is a file. Posts are posts, pages are posts, images are posts. The only thing differentiating them is their post_type. All posts are stored in the wp_posts table in the database.

WordPress uses an event-driven architecture. Its core philosophy is its hook-based system, which allows for extending the behavior of the website without modifying any of the core files. It's the whole reason why WordPress is so popular (because of the vast amount of third-party integrations allowed by its architecture).

It uses the Front Controller Pattern. All the requests (to non-static resources) are routed to the index.php file in the root directory. This file then "boots up" the entire WordPress machine.

WordPress File Structure

The structure outlined here strives to show some of the most important directories and files in WordPress. This knowledge will be useful for understanding the rest of this document. The list is by no means exhaustive.

- /
    - *.htaccess*
    - *index.php*
    - *wp-config.php*
    - *wp-load.php*
    - *wp-login.php*
    - **wp-admin/**
    - **wp-includes/**
    - **wp-content/**
        - **themes/**
            - **your-block-theme/**
                - *functions.php*
                - *style.css*
                - *theme.json*
                - **templates/**
                    - *index.html*

        - **parts/**
            - *header.html*
            - *footer.html*
    - **your-classic-theme/**
        - *functions.php*
        - *index.php*
        - *style.css*
- **plugins/**
- **mu-plugins/**
- **uploads/**
    - **2025/**
        - **07/**
            - *image.jpg*
- **languages/**

You have to remember one important rule. Never, under any circumstances, modify any files inside the wp-admin or wp-includes directories. All of your changes will be overwritten on the next WordPress update. Your playground is the wp-content directory.

WordPress Database Overview

This section is just a brief, high-level overview of the WordPress database schema. It's useful to gain some context about the database before we go further, but don't stress about it. The details of how the database is actually used will be interwoven in relevant topics. The high-level bullet points in the list below signify tables. The low-level ones signify some (not all) of the most important columns in each table.

- **wp_posts**
    - ID
    - post_author
    - post_date
    - post_content
    - post_title
    - post_excerpt
    - post_status
    - post_modified
    - post_parent
    - post_type
- **wp_postmeta**
    - meta_id
    - post_id
    - meta_key
    - meta_value
- **wp_terms**

    - term_id
    - name
    - slug
- **wp_term_taxonomy**
    - term_taxonomy_id
    - term_id
    - taxonomy
    - parent
- **wp_term_relationships**
    - object_id
    - term_taxonomy_id
- **wp_termmeta**
    - meta_id
    - term_id
    - meta_key
    - meta_value
- **wp_users**
    - ID
    - user_login
    - user_pass
    - user_nicename
    - user_email
    - display_name
- **wp_usermeta**
    - umeta_id
    - user_id
    - meta_key
    - meta_value
- **wp_options**
    - option_id
    - option_name
    - option_value
- **wp_comments**
    - comment_ID
    - comment_post_ID
    - comment_author
    - comment_date
    - comment_content
- **wp_commentmeta**
    - meta_id
    - comment_id
    - meta_key
    - meta_value
- **wp_links**

    - link_id
    - link_url
    - link_name

You know the schema, now it's time to understand it. The most important thing to remember about the database is that it's designed primarily for flexibility. WordPress has to work with thousands of third-party plugins and themes. It's the top priority that it doesn't limit the possibilities of extending functionalities with an unnecessarily constraining design.

The "wp_" prefix of every table can and should be modified during installation (for security reasons). You should almost never interact with the database directly. There are a gazillion core functions in WordPress that handle database interactions for you, either directly or indirectly. Some very direct examples are wp_insert_post(), wp_update_post_meta(), or get_post(). If you ever find yourself in need of custom SQL (e.g., if you use custom tables), you should use the global $wpdb object.

wp_posts is the heart of the database. Every post is stored in this table. The column that makes the "everything is a post" methodology possible is post_type. It stores the type of the post, e.g., post, page, attachment, nav_menu_item, etc.

wp_terms, wp_term_taxonomy, and wp_term_relationships are tables required for taxonomies to work. We'll dive deep into taxonomies later in this guide. All you need to know right now is that taxonomies are things like categories and tags. They let you create relationships and structure your posts.

wp_users stores the details of all users registered on the website. Some of the data are: login, password (hashed), email, display name, etc.

All meta tables (wp_postmeta, wp_termmeta, wp_usermeta, wp_commentmeta) are responsible for storing metadata associated with a given entry via a foreign key. You can see that all of those tables have basically the same columns. If you wanted to store some additional data (metadata) for a post (e.g., a product price), you would store it in the wp_postmeta table.

wp_options is pretty self-explanatory. It stores all the options for your website. If you change anything in the Settings menus, it'll most likely be stored in this table.

wp_comments stores comments left for posts.

wp_links is an obsolete table. It was used by the core Links Manager module, providing the blogroll functionality. It was basically just a set of links to other blogs. This functionality has been disabled by default since WordPress 3.5. The table is kept for backward compatibility.

Routing & Permalinks Structure

WordPress acknowledges 3 types of permalink structures:

- plain permalinks, e.g, http://example.com/?p=N, where N is the ID of the post,
- almost pretty permalinks, e.g, http://example.com/index.php/post-name/,
- pretty permalinks, e.g., http://example.com/post-name/

Pretty permalinks are naturally the most user-friendly and SEO-friendly. You can configure the type of permalink structure used in Settings -> Permalinks.

To be able to use pretty permalinks, you have to configure your web server accordingly. That means rewriting all requests (which do not match any static file or directory) to the index.php file. Here's what a typical .htaccess file in the root directory looks like:

```
RewriteEngine On
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

This file is generated automatically by WordPress if you change the permalink structure to one that requires it. The 'plain' and 'almost pretty' permalinks don't require a redirect as they are naturally routed to the index.php file. If you delete the .htaccess file, it'll be regenerated when the flush_rewrite_rules() function is called, such as when you visit the permalinks settings page.

The most important utility pages on WordPress do not require .htaccess, even if the pretty permalink structure is used. That's because they are all implemented to be accessed directly. The login page is by default /wp-login.php. That's just a php file. The admin dashboard's path is /wp-admin/, which is the name of the directory in the root folder. The index.php file inside /wp-admin/, which is called by default when the path is a directory, loads the admin dashboard. That's why you can see all the paths in the admin dashboard having a .php suffix (e.g., /wp-admin/plugins.php) - they are just php files.

The Request Lifecycle

Stage 1: index.php

This is the front controller. It's the first php file called on every request. It's incredibly simple; here's its entire code:

```
define( 'WP_USE_THEMES', true );
require __DIR__ . '/wp-blog-header.php';
```

The first line defines a constant that tells WordPress its goal is to display a theme. If you were writing a script that didn't need to load the theme, you could set this to false. The second line is the handoff, which initiates the rest of the functionality outlined below.

Stage 2: wp-blog-header.php

This file is also very simple. Here are the most significant lines of code in the file:

```
require_once __DIR__ . '/wp-load.php';
wp();
require_once ABSPATH . WPINC . '/template-loader.php';
```

wp-load.php is the main star. It loads the entire environment. The wp() function triggers parsing of the URL and the main database query. template-loader.php decides which template file to use.

Stage 3: wp-load.php

This file searches the current directory and its parent for the wp-config.php file. Why does it search the parent? Because then you can place the wp-config.php file, which contains confidential information, **outside** of your root directory. That makes it completely inaccessible to the web server, even if the server is (mis)configured to allow viewing php files as text. It's a hardening technique.

If wp-config.php is not present, a configuration process is initiated. That means loading a few crucial, bare-minimum files required to successfully redirect the user to the /wp-admin/setup-config.php file. The files loaded:

- **/wp-includes/version.php** - specifies some version variables, like the wp version or required php version.
- **/wp-includes/compat.php** - implementations of PHP functions either missing from older PHP versions or not included by default.
- **/wp-includes/load.php** - basic functions needed to load WordPress.
- **/wp-includes/functions.php** - more core WordPress procedural functions, many of which would not work at this stage. Crucially, it allows for internationalization to work.

In the typical case, the user will then get redirected to the /wp-admin/setup-config.php file - accessed directly without going through index.php. This file will then guide the user through the process of generating the wp-config.php file (write permissions for the root directory are required for it to work automatically; otherwise, the user is instructed to create the file manually).

Stage 4: wp-config.php

This is the configuration file. It sets up security keys, database credentials, debug configuration, and any other config (you can modify it if you need to). It defines this information by defining constants, e.g., define('DB_PASSWORD' 'secret_database_password'). An incomplete list of options that you can override, along with their default values, is available in /wp-includes/default-constants.php.

The last line of the wp-config.php file is:

```
require_once ABSPATH . 'wp-settings.php';
```

Stage 5: wp-settings.php

This is the final file in the chain of setting up the WordPress environment. Its responsibilities are:

- Include files from the /wp-includes/ directory providing core WordPress functionality.
- Set global settings and variables (e.g., default constants, language, timezone, etc.).
- Connect to the database and create the global $wpdb class.
- Load must-use plugins.
- Load active plugins.
- Load the active theme's functions.php file(s).
- Set up the current user's authentication and permissions.

After this file executes, the entire WordPress is "up & running" and loaded into memory, but it still doesn't know what to display and how to display it. The execution goes back to the wp-blog-header.php file.

Stage 6: wp() in wp-blog-header.php

This function has 2 main objectives: to parse the URL and query the database for the posts. The function itself is actually just a lean wrapper that delegates the work to an object of class WP ($wp). It calls its main() method, which in turn calls 2 very important methods:

- **$wp->parse_request($query_args)**
- **$wp->query_posts()**

parse_request() does exactly what you think it does - it parses the request URL. It interacts with the WP_Rewrite class and uses regex to match the URL path to the permalinks structure on the site (the one you set up in Settings -> Permalinks). It then sets those parsed fragments as variables, which will then be used to query the database for matching posts, e.g., a path /category/news/ would become [..., 'category_name' => 'news', …].

The query_posts() method is essentially a wrapper for the WP_Query class. Its goal is to fetch the requested posts from the database. After the method executes, the global $wp_query object will hold all the most important information, such as:

- **$wp_query->posts** - an array of WP_Post objects.
- **$wp_query->post_count** - the number of posts in the posts array.
- **$wp_query->found_posts** - the number of posts that matched the query (can be more than post_count if the result is paged).
- **Flags** - like $wp_query->is_single, $wp_query->is_category, etc.

For example, if the path is /category/news/, and the wp() function executes successfully, the result will be a populated $wp_query global object with the posts array containing 10 posts from the news category (by default). If the path is /post-name/, then the result will be a populated $wp_query object with the information and contents of that post, assuming the post exists.

Stage 7: template-loader.php

We now have the content to be displayed. It's time to display it. Execution flows to the last line of wp-blog-header.php, i.e.:

```
require_once ABSPATH . WPINC . '/template-loader.php';
```

The template-loader.php file is responsible for loading the appropriate template. It defines a static associative array, whose entries follow this pattern:

```
[
'is_embed' => 'get_embed_template',
'is_404' => 'get_404_template',
// ...
]
```

This array is then iterated over (if the WP_USE_THEMES constant defined in index.php is true). Both the key and the value are callbacks - names of functions to be called. The keys are very simple functions that check the flags on the $wp_query object and return their values. Once the key callback returns true, the value callback is called, and the loop is broken out of. The entries in the array are ordered from most to least specific.

The value callbacks are functions responsible for actually finding the correct template to include. Let's take get_category_template(). It looks for files in the current theme that match WordPress's template hierarchy, e.g., "category-news.php", "category.php", etc. It then includes the most specific one found. The template hierarchy is a topic worth its own section.

If none of the callbacks return true, WordPress defaults to displaying the index.php file (the one in the theme directory, not the front controller).

No more files are called by WordPress. Its job is done. It has set up the entire environment and called the appropriate template defined by the theme used on the site. That's where the responsibility shifts from WordPress to the website/theme developer.

Hooks

As previously noted, hooks are the most significant part of WordPress's architecture. They provide a way to interact with/modify code in predefined spots. They are the foundation for themes and plugins to interact with the WordPress core. They work by registering callbacks to be executed when the hook is called. There are 2 types of hooks - actions and filters.

Actions

Actions provide a way to run code at specific points in the execution. Here's how you hook your function to an action:

add_action($hook_name, $callback, $priority = 10, $accepted_args = 1);

$hook_name is the name of the action you're hooking into. WordPress comes with hundreds of actions in the core itself, and you have to choose the appropriate one based on where you want your function to run. $callback is the name of your callback function.

You can specify the relative priority of your function with the $priority argument. The lower the priority, the earlier your callback will be run. Theoretically, there is no lower or higher bound on priority. That being said, it's best practice to only use priorities between 1 and 20. Using a negative priority is rare, but can be useful in advanced scenarios, where you have to guarantee your function runs before all others (most functions will keep the default priority of 10 anyway).

The $accepted_args argument indicates the number of parameters you want the action to pass to your callback function. A hook can, and often does, pass some data to the callback. Let's look at the 'save_post' action. It passes 3 parameters - $post_id, $post (WP_Post object), and $updated (bool). If you want to use the last 2, you'd have to explicitly set $accepted_args = 3, and make your callback function accept 3 parameters.

Thankfully, php doesn't care if you pass too many parameters to a function, which is why the default value of 1 doesn't cause errors if your callback doesn't accept any parameters. How do you know what parameters are passed by a given hook? You look at the official WordPress Hooks Reference (or at the source code).

An example use of add_action():

```
function callback_on_save( $post_id, $post, $updated ) {
```

```
    // Some code
}
add_action( 'save_post', 'callback_on_save', 10, 3);
```

do_action($hook_name, $args…);

The do_action() function executes all the callbacks hooked to the specified action in the correct order of priority. The parameters to be passed to callbacks are specified after the hook name. Here's what the code for 'save_post' looks like:

```
// Define $post_id, $post, and $updated
do_action( 'save_post', $post_id, $post, $updated );
```

The entire WordPress code has hundreds of those do_action() calls scattered across the entire core. They are strategically placed to let plugin and theme developers do something before or after something significant happens.

You can also define your custom actions with this function. The only thing you need to do is use it with a unique name, like do_action('my_theme_action'). It doesn't require any sophisticated registering of a hook name or anything. Just call do_action() and any callback that has been registered with this hook name will get executed in that place.

Filters

Filters provide a way to modify some piece of data. Unlike actions, filters are supposed to work in an isolated manner and should not have any side effects, such as affecting global variables or throwing exceptions.

add_filter($hook_name, $callback, $priority = 10, $accepted_args = 1);

The idea is the same as for add_action(). As a matter of fact, you can see the arguments are exactly the same. It's what happens with the callback that differs.

The purpose of the filter is to pass some data to the callback, and then receive a "filtered" version of that data in response. That means all of the filters will pass in at least one parameter. This is where the default value of $accepted_args comes in handy. Filters can pass more parameters, just like actions.

The callback has to return a value. The returned value becomes the new, updated value for the variable being filtered. This modified value is then passed to the next filter in the order of priority.

An example use of add_filter():

```php
function add_exclamation_mark_to_title( $title ) {
	return $title . '!';
}
add_filter( 'the_title', 'add_exclamation_mark_to_title' );
```

apply_filter($hook_name, $value, $args…);

This is the do_action() equivalent for filters. It executes the callbacks hooked to $hook_name, passing in the original $value and optional additional $args. Just as with actions, you can define your own filter by calling apply_filter() with a unique hook name.

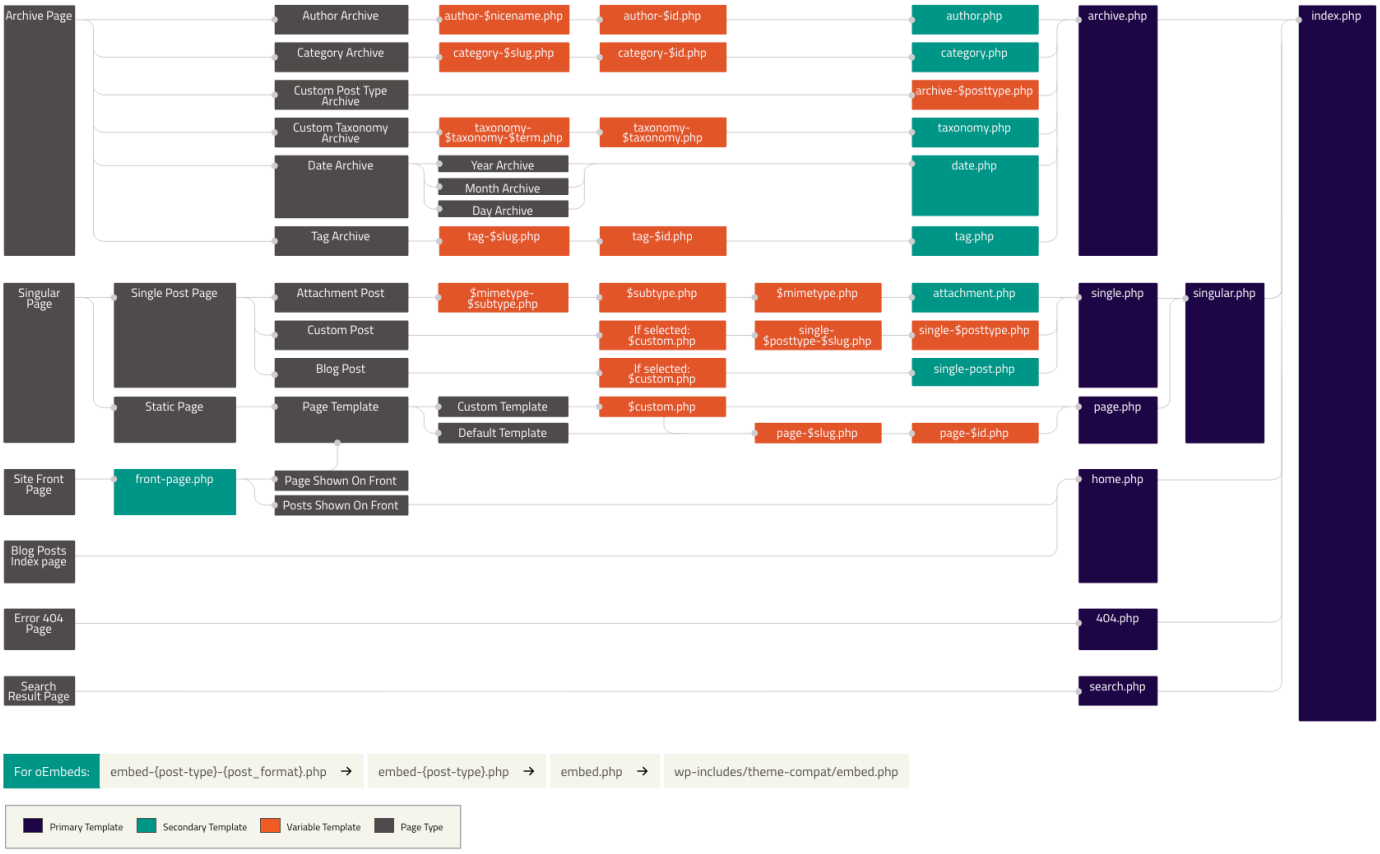
Removing Actions And Filters

Sometimes you might want to remove an action or filter registered by a theme, plugin, or even WordPress core itself. You can do that with remove_action() or remove_filter(). The $callback and $priority passed to those functions have to be the same as those used for registering the action/filter. You can only remove the action/filter after it is added - the order of execution matters. Here's how you could remove the previously registered functions:

```php
function remove_custom_callbacks() {
	remove_action( 'save_post', 'callback_on_save', 10 );
	remove_filter( 'the_title', 'add_exclamation_mark_to_title' );
}
// use after_setup_theme to guarantee execution after the original hooks have been added
add_action( 'after_setup_theme', 'remove_custom_callbacks' );
```

Template Hierarchy

The template hierarchy specifies the order in which templates are checked for existence and loaded. Templates are just files (.php for a classic theme and .html for a block theme). They are kind of like the "View" in the Model View Controller pattern (even though WordPress does not utilize MVC). For every post type, WordPress looks for pre-defined file names in the active theme directory. It starts with the most specific one and loads the first one found. Different post types have different hierarchies. More information on the template hierarchy can be found in the WordPress Template Hierarchy Documentation. The infographic below shows the hierarchy visually.



Source: [WordPress Template Hierarchy Documentation]

Front Page & Home Page

The topic of the front page and the home page in WordPress is very confusing. Let's get the definitions straight first.

Front page - the page displayed for the root URL of your website (/), e.g., https://example.com/.

Home page - the page responsible for displaying all posts. Yes, this is stupid. Why isn't it called "Posts archive" or "Posts index"? Well, because of WordPress's history. Back in the day, the front page and the home page were the same thing - the only thing you could display on the front page was posts.

The front and home pages can be set up in Settings > Reading > "Your homepage displays". There are 2 options:

- Your latest posts (default)
- A static page

The default behavior means the home page **is** the front page - the posts index is displayed for the root URL.

The static page option gets weird. It lets you choose a page (a post with post_type = page) for the front and home pages:

- Homepage (front page)
- Posts page (home page)

See? The names are normal here. The homepage is our front page, and the posts page is our home page. Don't get fooled though, this is just to make the experience in the admin panel better. All the functions in WordPress still follow the old pattern, with is_home() instead of is_posts_page() (which doesn't exist).

Template Hierarchy With 'Your latest posts' Option Checked

In this scenario, the front page is also the home page. That means when the user visits your base URL (https://example.com/), $wp_query will hold, by default, the 10 most recent blog posts. The template hierarchy is as follows:

1. front-page.php
2. home.php
3. index.php

Template Hierarchy With 'A static page' Option Checked

This gets more complicated and confusing again. When using this option, WordPress treats these 2 with completely separate hierarchies.

The 'Homepage' option lets you select which page should be displayed when a user visits the root URL. The template hierarchy is as follows:

1. front-page.php
2. Custom page template (if assigned in the chosen page's editor)
3. page-$slug.php
4. page-$id.php
5. page.php
6. singular.php
7. index.php

You don't know it yet, but the hierarchy right after front-page.php is the hierarchy of a page. Notice that the front-page.php file has priority over the template of the chosen page. In that case, $wp_query will contain the contents of that page, but front-page.php will be used as the template.

The 'Posts page' option lets you select which page should display blog posts (i.e., populate $wp_query with, by default, 10 recent posts). The template hierarchy is as follows:

1. home.php
2. index.php

Noticed something surprising? The template of the chosen page is not here! As a matter of fact, the only role of the page you choose is to define the URL (and metadata) for the blog page. The content you put in the editor of this page will never be displayed, as $wp_query is already occupied with posts.

Post Of Any Type Except Attachment (blog post, custom post type, etc.)

The template hierarchy of a single post:

1. Custom page template (if selected)
2. single-$post_type-$slug.php
3. single-$post_type.php
4. single.php
5. singular.php
6. index.php

Attachment

An attachment (e.g., an image) gets a permalink in WordPress, just like any other post would. This permalink is not the direct link to the file on the server, i.e., it doesn't end with .jpg, .png, .pdf, etc. It's literally a page on your website just for displaying your attachment. It's a "custom" URL, and as such, goes through the WordPress request lifecycle.

This feature is not widely used anymore and sometimes even considered undesirable because these pages usually have thin content (which is a negative SEO factor). I've also tried playing with it while writing this and have been hit with some very weird behavior, like WordPress redirecting me to the file URL even though the attachment template existed.

Anyway, here's the (theoretical) template hierarchy for the attachment post type if you ever need it:

1. $mime-$subtype.php
2. $subtype.php
3. $mime.php
4. attachment.php
5. single-attachment.php
6. single.php
7. singular.php
8. index.php

Page

A single static page template hierarchy:

1. Custom page template (if selected)

2. page-$slug.php
3. page-$id.php
4. page.php
5. singular.php
6. index.php

Category Archive

The template hierarchy for the page displaying posts of a given category:

1. category-$slug.php
2. category-$id.php
3. category.php
4. archive.php
5. index.php

Tag Archive

Template hierarchy for the page displaying posts with a given tag:

1. tag-$slug.php
2. tag-$id.php
3. tag.php
4. archive.php
5. index.php

Custom Post Type Archive

Template hierarchy:

1. archive-$post_type.php
2. archive.php
3. index.php

Custom Taxonomy Archive

This applies to any taxonomy that is not a built-in category or tag. Template hierarchy:

1. taxonomy-$taxonomy-$term.php
2. taxonomy-$taxonomy.php
3. taxonomy.php
4. archive.php
5. index.php

Author Archive

This archive displays posts from the given author. Template hierarchy:

1. author-$nicename.php

2. author-$id.php
3. author.php
4. archive.php
5. index.php

Date Archive

This archive displays posts from the given date (e.g., example.com/2023/, example.com/2023/07/, example.com/2023/07/01/, etc.)

1. date.php
2. archive.php
3. index.php

The 404 Page

Template hierarchy:

1. 404.php
2. index.php

Custom Post Types

We've already established that everything in WordPress is a post. They differ by their post type. There are over a dozen post types available in WordPress by default (as of July 2025), but the real magic comes from custom post types. Here's a list of some of the most important default WordPress post types:

- Posts
- Pages
- Attachments
- Revisions
- Navigation Menus
- Block templates
- Block template parts

Let's assume we're creating a website for a library. We need a way to display books. The best way to achieve that would be to create a custom post type 'book'. But why? Why not just use the default post with a 'Books' category? Here's why custom post types are so amazing:

- **A separate admin tab** - books will get their own tab in the admin menu instead of being mixed with blog posts.
- **Custom fields** - a book needs different data than a simple post. It has the author, release date, number of pages, number of copies sold, etc. You can add custom meta boxes specifically for inputting content like that. You can even add image galleries or date pickers. You can then use this data in the book's template.

- **More control over templates** - if you looked at the post's template hierarchy, you would see that there's no way for us to create a separate template for posts of different categories. But you can create a template for a different post type. In our case, we could create a single-book.php template, which would display the single book page (we don't want it to look like a simple post, do we?).
- **Semantic correctness** - a book is not a blog post. You wouldn't treat pages with books the same way you treat blog posts. They are semantically different, and as such, they should be structurally separate. The most obvious example is the posts archive, which you would create with the home.php file. Do you really want books to show up among your posts?
- **Custom permalinks structure** - a separate post type gets its own permalinks structure. Instead of having /category/books/book-title, you can have /books/book-title.

So how do you register a custom post type? You use the register_post_type() function. Let's register our 'book' post type:

```
function register_book_post_type() {
    $labels = [
        'name'          => 'Books',
        'singular_name' => 'Book',
        'menu_name'     => 'Books',
        'add_new_item'  => 'Add New Book',
        'edit_item'     => 'Edit Book',
        // ... other labels
    ];

    $args = [
        'labels' => $labels,
        'public' => true,
        'hierarchical' => false,
        'has_archive' => true,
        'menu_icon' => 'dashicons-book',
        'supports' => [ 'title', 'editor', 'thumbnail', 'excerpt', 'custom-fields' ],
        'taxonomies' => [ 'genre' ],
        'rewrite' => [ 'slug' => 'books' ],
    ];

    // The first argument is the unique name
    // (max 20 chars, no spaces/caps, should be prefixed with a
    // unique theme/plugin identifier to avoid conflicts - 'thm' here for "theme")
    register_post_type( 'thm_book', $args );
}
add_action( 'init', 'register_book_post_type' );
```

This list of arguments is not exhaustive, and I'm not going to explain them. RTFM.

Your custom post types should be registered in a plugin, not a theme, as themes are supposed to be responsible for presentation only (more on that later). This rule is often bent, especially if the website in question is not likely to be migrated to a different theme. You will see custom post types registered in functions.php a lot.

Custom Fields

Custom fields are just post metadata. There's nothing more to it. This metadata is stored in the wp_postmeta table in the database. It's a one-to-many relationship, meaning a single post can have multiple custom fields. This provides nearly limitless possibilities. It serves as a more structured content, allowing you to store information such as product price, product stock, the number of pages a book has, or even what type of layout should be used on the page.

There's an important distinction to be made between the 2 main terms - custom fields and meta boxes. A custom field is the more abstract concept of metadata stored for a post. A meta box is the actual HTML form element rendered in the post editor. It's the input allowing the post author to fill the custom field with data.

You could add metadata to your posts with PHP using functions like add_post_meta() or update_post_meta(). This would create a key-value pair in the wp_postmeta table with whatever key and value you supplied to the function. That would be a valid way of adding metadata if you wanted to control it using only code. But that's usually not the case. Instead, you want the post authors to be able to specify the data for each post. That's when you need to create a meta box.

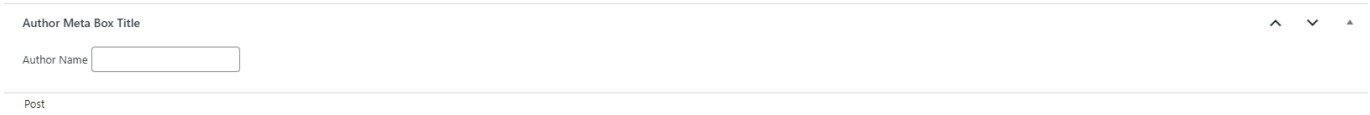
Creating A Custom Field & Saving The Metadata

To create a custom meta box, you have to use the add_meta_box() function. You specify its ID, title, content callback, and the post type(s) for which the meta box is to be displayed. The callback is responsible for rendering the HTML of the meta box. Let's add an "Author Name" meta box to our book post type:

```
// DON'T USE IN PRODUCTION (NOT SECURE)
function thm_book_author_name_html( $post ) {
      // You'll understand this later
      $value = get_post_meta( $post->ID, '_thm_author_name', true );

      ?>
      <label for="thm_author_name_field">Author Name</label>
      <input type="text" name="thm_author_name_field" id="thm_author_name_field"
```

```
value="<?php echo $value ?>">
	<?php
}

function thm_add_custom_box() {
	add_meta_box(
		'thm_book_author_name', // Unique ID
		'Author Meta Box Title', // Box title
		'thm_book_author_name_html', // Content callback, must be of type callable
		'book', // Post type
	);
}
add_action( 'add_meta_boxes', 'thm_add_custom_box' );
```

And here's what it looks like in the post editor. It shows up at the very bottom of the page. You could also reorder meta boxes with the up and down arrows if you had more of them, or place them in the Inspector sidebar on the right. You could also place multiple <input> elements under the same meta box.



That's working, but if you tried to input the author's name and saved the post, you would see it didn't get saved. And no wonder - we haven't written any code to save it yet! To do that, we need to create a PHP function and hook it into some action executed after the post is saved (usually save_post).

The input to the meta box is sent along with all the other post data in a POST request when the user saves the post. There's no special treatment for meta boxes. It works as if the entire post were one huge <form> element. This means we have to dig through the payload, read the data, and save it in the database.

```
// DON'T USE IN PRODUCTION (NOT SECURE)
function thm_save_cf_book_author_name( $post_id ) {
	if ( array_key_exists( 'thm_author_name_field', $_POST ) ) {
		update_post_meta(
			$post_id,
			'_thm_author_name',
			$_POST['thm_author_name_field']
		);
	}
```

```
}
add_action( 'save_post', 'thm_save_cf_book_author_name' );
```

This function searches the global $_POST array for the key 'thm_author_name_field' (which is the name of our meta box's input element). It then updates the entry in wp_postmeta with key '_thm_author_name' and the user-supplied value for the saved post.

The function update_post_meta() is usually preferred over add_post_meta() as it updates the value if the key already exists, and creates it if the key is not present. add_post_meta() always adds a new entry. This means you would have 10 values associated with '_thm_author_name' if you saved the post 10 times (yes, a single meta key can have multiple meta values).

The above example is theoretically all you need to add a custom field to a post. However, you should not use it in production. It lacks sanitization and other security checks. It's also incredibly simple, as we only needed a text input field. But what if you needed a date picker or an image gallery? That's where practice meets theory.

The Reality Of Custom Fields

In practice, coding custom fields (and custom meta boxes) is done only by theme and plugin developers. If you were to create a website for a client, you'd almost never write your meta boxes. It's too tedious, complicated, and error-prone. And most importantly - it's reinventing the wheel.

There are many plugins that solve this exact problem. One of the most loved and popular is Advanced Custom Fields (ACF). It's a commercial plugin, but you'll see it mentioned by professional WordPress developers all the time. As a matter of fact, it's so popular that in 2024 it got hijacked by WordPress's mother company Automattic, when Matt Mullenweg made an idiot of himself trying to extort $32M from WP Engine.

This is not a tutorial on ACF, so I'm not going to explain it. The bottom line is that it takes care of creating those custom fields for you, with over 30 native field types (meta box inputs) available, including advanced ones like files, images, date pickers, etc.

Using Metadata In Templates

You've created your custom fields. Now you want to use or display them in your templates. How do you do that? By calling get_post_meta($post_id, $key, $single). $post_id is self-explanatory. $key is the meta_key you want to retrieve. $single is a boolean value. If true, it returns only the first value associated with the specified key. If false (default), it returns an array of values (no matter how many there are).

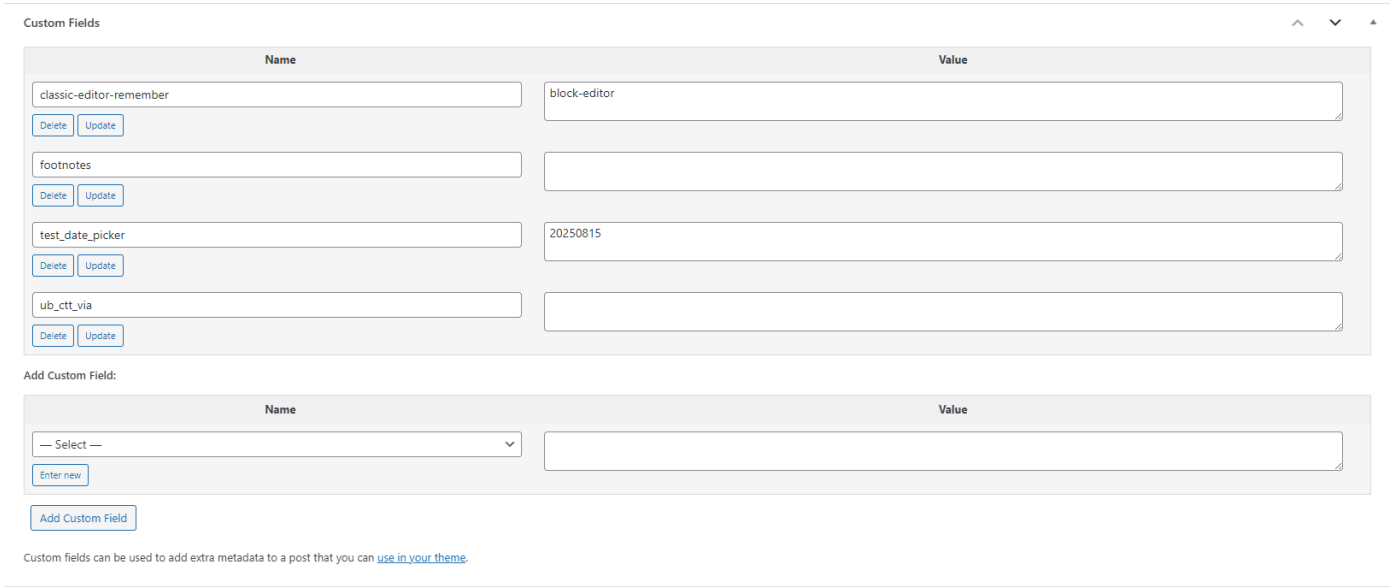
If you were using a plugin for custom fields, you'd usually use its specific functions. For example, ACF provides the get_field() and the_field() functions. These functions take care of formatting the data for you. If you were to do get_post_meta() directly on a date picker field, you'd get a value like '20250809'.

The Native Custom Fields Tab In The Editor

Throughout this entire section, we've been talking about creating custom meta boxes when adding custom fields. You might be surprised to hear that the WordPress editor has a native way of adding and modifying metadata. I'm covering it at the very end because the truth is that it's very obscure and not user-friendly.

To show the "Custom Fields" tab in the Block Editor, you have to navigate to the three dots > Preferences, and under General > Advanced, switch "Custom Fields" on. After refreshing the page, you'll see a "Custom Fields" area appear at the bottom of the editor.

This area displays all non-hidden metadata for the post. The problem is that it's only a text field, and there are no other field types. The user can mess with the meta_key, which is not good, and the general user experience is horrible. This option is not viable for websites where the person using it is not a developer or where there are advanced fields needed. You can see what I'm talking about in the screenshot below.



As you can see, there are a couple of fields added for this post. The classic-editor-remember was added by the Classic Editor plugin to save the editor preference for this post. The test_date_picker was added by me with ACF, and you can see the value format I was talking about. By the way, I've disabled ACF before taking that screenshot, and you can see the data in the wp_postmeta table is still there (as it should).

Why did I say "it displays all non-hidden metadata"? Are there hidden metadata? The answer is yes. Look back at my code example from the beginning of this section for our Author Name custom field. Then look back at the screenshot above. It's not there. I'll give you a moment to figure out why… Ready?

Metadata for which the key starts with an underscore is considered hidden. It's like dotfiles in Linux. These fields are hidden from the Custom Fields area in the editor (and from the the_meta() function, but that's deprecated anyway).

Taxonomies & Terms

"Taxonomy is a practice and science concerned with classification or categorization." ~ Wikipedia.

A post type defines a type of content, while a taxonomy defines a way to categorize that content. For example, you could have a taxonomy "Genre" for a book post type. Terms are just single options in the taxonomy. For genre, you could have terms like "fantasy", "sci-fi", "business", etc.

The generic post has, by default, 2 taxonomies - category and tag. Category is a hierarchical taxonomy. This means terms can have parent-child relationships. A "Technology" category can have subcategories like "Frontend", "Backend", etc. Tag is a non-hierarchical taxonomy. All terms (tags) are on the same level.

Here's how to register a "Genre" taxonomy for our book post type:

```
function register_genre_taxonomy() {
    $labels = [
        'name'               => 'Genres',
        'singular_name'      => 'Genre',
        'search_items'       => 'Search Genres',
        'all_items'          => 'All Genres',
        'edit_item'          => 'Edit Genre',
        'update_item'        => 'Update Genre',
        'add_new_item'       => 'Add New Genre',
        'new_item_name'      => 'New Genre Name',
        'menu_name'          => 'Genres',
    ];

    $args = [
        'labels'         => $labels,
```

```
        'hierarchical'   => true,
        'public'         => true,
        'show_ui'        => true,
        'rewrite'        => [ 'slug' => 'genre' ],
    ];

    // 1st arg: The unique taxonomy name.
    // 2nd arg: The post type(s) it applies to.
    // 3rd arg: The arguments array.
    register_taxonomy( 'thm_genre', [ 'thm_book' ], $args );
}
add_action( 'init', 'register_genre_taxonomy' );
```

Just as with custom post types, the list of arguments isn't exhaustive, RTFM. After this taxonomy is successfully registered, a "Genres" sub-menu will appear under the "Books" tab in the admin panel. You'll be able to add terms there. Yes, you don't register terms in code. You create them in the admin panel, just like you would create a post. Then, when editing a "book" post, you'll see a meta box letting you select the genres you want to assign this book to.

You can also connect an existing taxonomy to a post type. A primary use case is making the built-in category or tag taxonomies available to custom post types. You'd use the register_taxonomy_for_object_type($taxonomy, $post_type) function.

Taxonomies In The Database

The way taxonomies and terms are stored in the database is a little complex, but it's worth understanding. The structure is based on 3 tables: wp_terms, wp_term_taxonomy, and wp_term_relationships.

wp_terms stores all of the terms. This table has only 3 important columns: term_id, name, and slug. It stores no information on the relationships between those terms and taxonomies. If we were to add the term Fantasy to our Genre taxonomy, its name would be stored here along with its ID (which is just an auto increment field).

wp_term_taxonomy connects terms with taxonomies. The 3 most important columns are term_taxonomy_id (auto increment), term_id, and taxonomy. term_id stores the foreign key from wp_terms. It's the unique ID (term_id) of the term. The taxonomy column stores the name of the taxonomy as registered in code. This would be "thm_genre".

Keep in mind, no data about the taxonomy itself is stored in the database. It's all defined in code when WordPress loads (in that case - in our register_genre_taxonomy() function). There's also a parent column which is used if your taxonomy is hierarchical (like category is) and the term

has a parent. It's the term_taxonomy_id of the parent term (and 0 if the term doesn't have a parent).

wp_term_relationships is the glue between posts and terms. The only 2 significant columns are object_id and term_taxonomy_id. object_id is the ID of the post (post_id from the wp_posts table). term_taxonomy_id is a foreign key from the wp_term_taxonomy table. It connects the post with the appropriate term.

Media Library & Files

Media in WordPress are managed using the Media Library. You can get there by heading to the 'Media' menu on the left or to the /wp-admin/upload.php URI. In there, you can drop in all kinds of files - images, PDFs, binary files, and even videos. You can delete files as well. You can also modify files' titles, captions, descriptions, and alternative texts.

Every file is a post, feeding into the "Everything is a post" mentality. Its post type is 'attachment'. All of these files are stored in the /wp-content/uploads/ directory. By default, they are organized in /yyyy/mm/ subdirectories. This means that if it's August 2025, and you upload a file, it'll be placed in /wp-content/uploads/2025/08.

You can change this behavior by unchecking the "Organize my uploads into month- and year-based folders" checkbox in Settings > Media. If you do that, all of your files will be uploaded straight to the /wp-content/uploads/ directory. Keep in mind that the directory can become very large very fast, especially if you have a media-heavy website.

Image Sizes

One of the most powerful features of WordPress's media system is image sizes. When you upload an image in the Media Library, WordPress automatically generates various different registered sizes of that image. Let's look at it in more detail.

The add_image_size($name, $width, $height, $crop) function allows you to register a new image size. Every image uploaded from then on will have an additional version file generated following this pattern: {original_slug}-{width}-{height}.{file_extension}. How the image will be scaled depends on the $crop argument. You can either not crop at all (default), crop from the center, or crop from a specified position (left/center/right and top/center/bottom).

Let's assume you registered a 500x500 image size without cropping, and you uploaded a "cat.png" file with a native size of 1000x800. The additional generated file will be named "cat-500x400.png". You can see the image got scaled without distorting the aspect ratio.

All built-in WordPress functions responsible for getting/displaying images accept a size argument. Let's say you registered your custom size with the name "square-medium". You can

call wp_get_attachment_image($attachment_id, 'square-medium'). This will return the HTML of an img element linking to the 500x400 version.

There are a few image sizes registered by default. Some of them you can modify manually in Settings -> Media. Those are: thumbnail (150x150), medium (300x300), and large (1024x1024). Thumbnail is cropped by default, while all the other sizes are scaled proportionally. There's also a default medium-large size (768x768), which, interestingly enough, cannot be modified in settings.

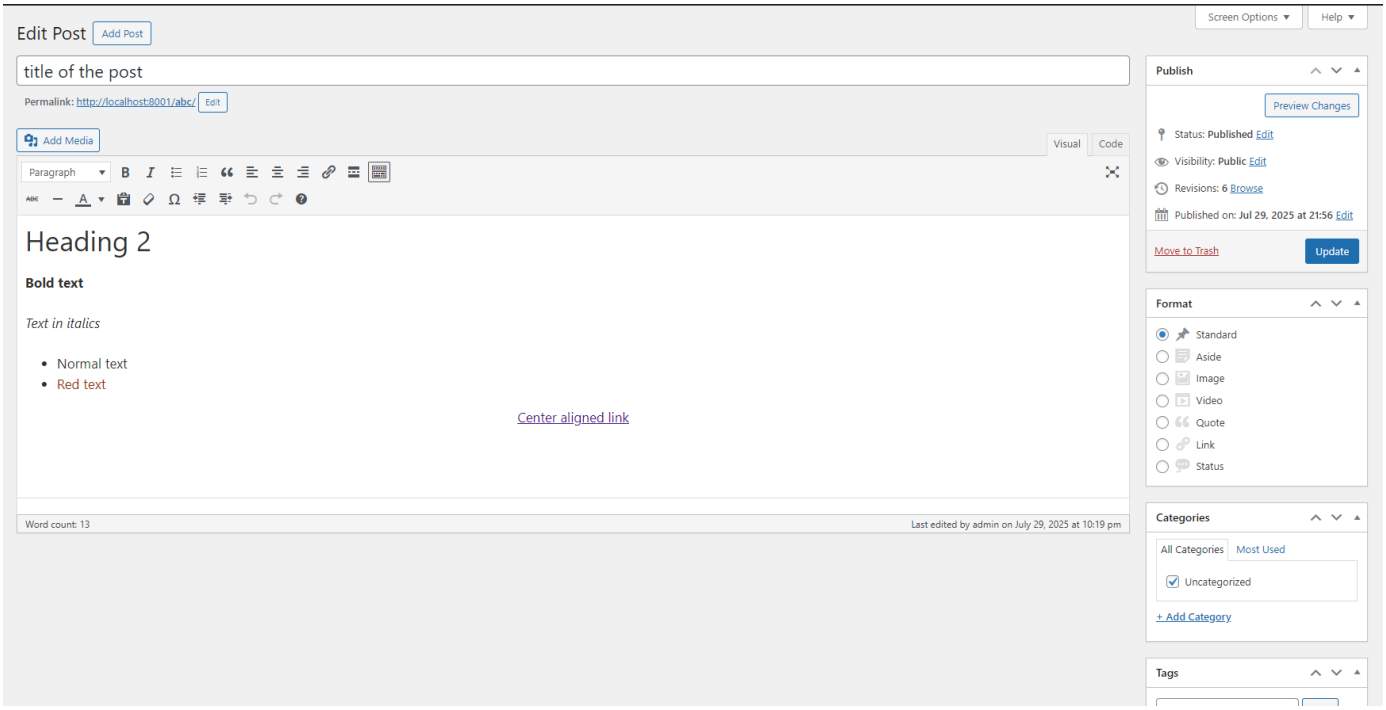
Image sizes are not separate posts. Information about them is stored in the wp_postmeta table. Every attachment post has a "_wp_attachment_metadata" entry. This entry stores a lot of the information about the image, including all of the generated sizes and the file's innate metadata.

PS: Images added in the past will not be regenerated after registering a new image size. You'd have to use a plugin for that.

The Classic Editor

Every post (and every page) has an editor. It's the place where you can write content to be displayed for said post, change its title, permalink, terms associated with it, etc. The classic editor, as described below, was introduced into WordPress in 2005. It was a big improvement as it provided some WYSIWYG capacity that WordPress didn't offer up to that point. This version of the editor was loved by many until it was pushed out by the block editor in 2018.

The classic editor is based on an open-source JavaScript library called TinyMCE. It provided an experience comparable to creating a document in Word. It had basically just one <textarea> with different controls, letting you add a bullet list, make the text bold, set a line to be a heading level 2, and much more. The visible content was then translated into HTML, which you could see if you switched your active tab from "Visual" to "Text" (or "Code" in modern versions of the editor). Take a look at what the classic editor looked like below.



Unfortunately, this editor was very limiting. The entire content of this text area was stored as a single blob of HTML in the post_content column in the database. If you wanted to add anything more complex than simple text and media, you had to use shortcodes (e.g., [my_gallery id="123"]). These were placeholders, usually provided by plugins, that would be parsed and replaced with some HTML on the frontend.

A simple text editing experience like that was fine in 2005 when most websites were just text anyway, but as the web evolved, so did the requirements of the editor. This manifested itself with a booming rise of page builders, like Elementor or Divi, which often replaced the native editor altogether. WordPress tried to respond to that demand in 2018 by replacing the classic editor with the block editor. Today, you can only use the classic editor if you download a plugin for it.

Shortcodes

As already noted, shortcodes are a way of including complex and/or dynamic content in a post using the Classic Editor. They are a very old feature, added in WordPress 2.5. The basic idea is very simple - you write a PHP function, register it with a name, and then someone uses that name in square brackets inside the editor. Let's see what it really looks like.

A Hello World Shortcode

```
function hello_world_shortcode( $atts ) {
```

```
	return '<p>Hello, World!</p>';
}
add_shortcode( 'hello_world', 'hello_world_shortcode' );
```

That's it. Now you can write [hello_world] in your editor and it'll be replaced with "Hello, World!". Notice that you should return the content (HTML) as a string, not echo it. This is the simplest possible version of a shortcode. Let's look at one that utilizes attributes.

Shortcode With Attributes

```
function hello_world_shortcode( $atts ) {
	$a = shortcode_atts( array(
		'world' => 'World',
	), $atts );

	return '<p>Hello, ' . $a['world'] . '!</p>';
}
add_shortcode( 'hello_world', 'hello_world_shortcode' );
```

This is a little more interesting, isn't it? First of all, you'd use it like: [hello_world world="World"] to render "Hello, World!". But we're using an attribute, which means we can do [hello_world world="WordPress"] to render "Hello, WordPress!". You can see, attributes provide a way for the user to pass data to the callback function (which can introduce XSS vulnerabilities if the data is rendered like here - sanitize your inputs!).

But what is this mysterious shortcode_atts() function? To understand its significance, we have to understand the behavior of the $atts array.

First of all, any attribute can be passed to the shortcode by the user, and it'll be present in $atts. This means if the user did [hello_world abcd="rogue attribute"], this would make the $atts array contain an 'abcd' key-value pair. This is weird, as we don't use this attribute in our callback and we've never said we want it.

You can see the second major point in the previous example as well. Notice that there's no 'world' attribute. The user omitted a crucial attribute we're using in the code. If you were to use the $atts array directly, you'd get a PHP error for trying to access an undefined key ($atts['world']).

The shortcode_atts() function solves both of these problems. The first argument is a new associative array. This array defines the expected attributes along with their default values. If an attribute specified in this array is not present in $atts, it'll be added. The second argument is the $atts array. In addition to adding attributes with their default values, this function filters out

unanticipated attributes as well. Any attribute present in $atts but absent from the first array will not be included in the returned array (the original $atts array is not modified).

Attributes are pretty powerful, but there's another way. It's using enclosed content.

Enclosing Shortcode

```
function hello_world_shortcode( $atts, $content = null ) {
      return "<p>Hello, $content!</p>";
}
add_shortcode( 'hello_world', 'hello_world_shortcode' );
```

You would then use this shortcode in the editor like: [hello_world]World[/hello_world].

There's also a third, rarely used argument passed to the callback function. It's $tag. It contains the name of the shortcode used to call this function. In our case, it'd be 'hello_world'. This might be useful if you're using the same callback function for multiple different shortcodes.

do_shortcode()

The do_shortcode($content) function is responsible for parsing the $content and replacing any registered shortcodes with their returned HTML. Keep in mind, a shortcode needs to be registered with add_shortcode() in order to be rendered. This can often be very problematic, where a plugin providing the shortcode is disabled, and the text (e.g., "[hello_world]") is displayed instead of the shortcode's output.

This function is hooked to the the_content filter with priority 11 by default. This filter passes the post's content. Priority 11 ensures it's run after all filters with default priority (10) have already run, as they may modify/add shortcodes to the contents of the post. After this function runs, all the registered shortcodes that are properly used in the post's contents are rendered.

While shortcodes are powerful, the user can't see what they'll look like until they preview the rendered page. This makes them less than ideal, and it's exactly what blocks and the Block Editor were created to solve.

Blocks & The Block Editor

This is perhaps the most important section when it comes to understanding modern WordPress. Blocks and the block editor have fundamentally changed the way content is created and thought of in WordPress.

Blocks

A block is an abstract component used on the website. Let's say you want to have a countdown on your website. This countdown may be a block. You can then place it in your templates, on your pages, or in your post's content. A button is a block. An image is a block. A paragraph and a heading are blocks. All blocks usually give you some attributes to customize in the editor (font size, spacing, text content, etc.).

Blocks are just elements that render some pre-defined HTML on the frontend. You get to decide what that is. It can be something complex with 15 different HTML elements, or it can be something very simple (like the core abstractions over foundational elements like <p>). Here's what the core paragraph block looks like:

```
<!-- wp:paragraph -->
<p>Welcome to WordPress. This is your first post. Edit or delete it, then
start writing!</p>
<!-- /wp:paragraph -->
```

And here's a block from the Ultimate Blocks plugin:

```
<!-- wp:ub/click-to-tweet
{"blockID":"5545214f-81c0-4e12-8325-6f0c79da171c","ubTweet":"Content to
tweet","padding":{"top":"0","bottom":"0"}} /-->
```

This is what gets saved in the post_content column in the database when you're using the block editor. When WordPress renders the content to be displayed on the frontend, it doesn't just return whatever is stored in the database. It parses the HTML comments (text between arrows) and renders any blocks it finds. These block delimiters transform the pseudo-unstructured blob of HTML into a structured tree of components.

This lets you create reusable, consistent, high-level components, which you can then structure your content with. Blocks are like shortcodes, except they have a much better user experience and get rendered in the editor.

In some way, you could say blocks are like HTML, but at a higher level of abstraction. They also have to get parsed, and just like HTML, they have attributes that modify their content and/or behavior. They are an abstract layer over HTML. This improves the user experience for a typical user, but it is more constraining than writing the HTML by hand. A trade-off that's beneficial in 95% of cases.

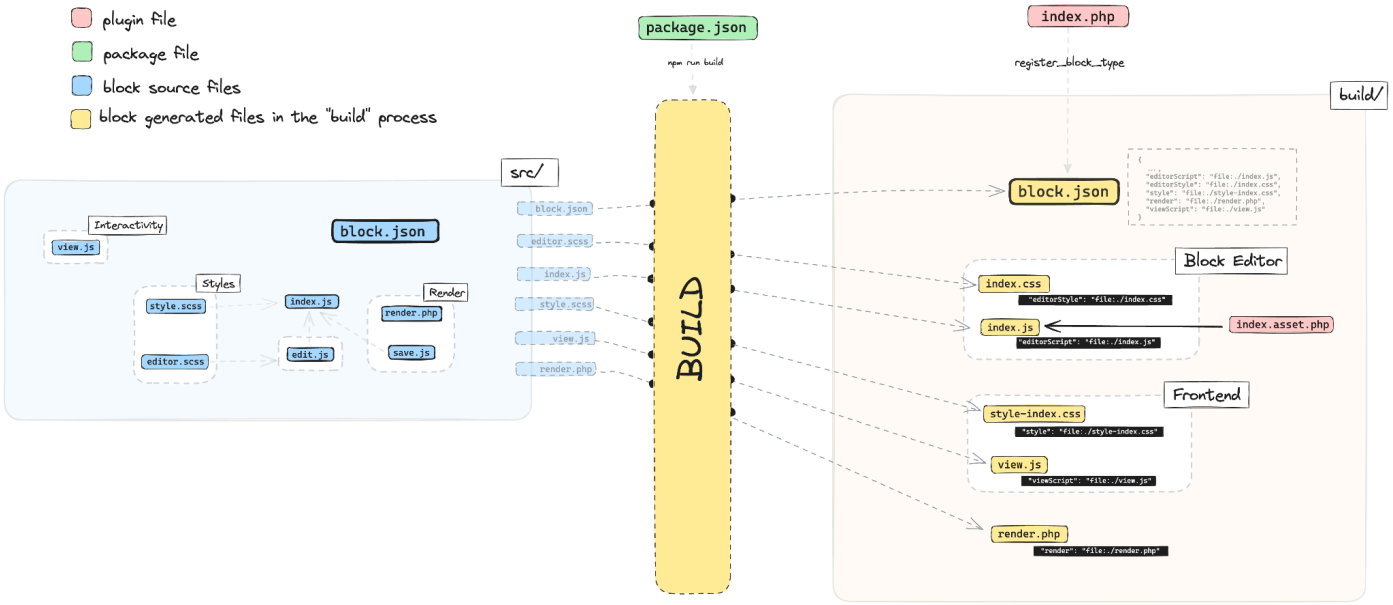
A Technical Deep Dive Into Blocks

This section could be an entire document. I'll try to explain how blocks work and are created, without going into too much detail. Read the official [WordPress Block Editor Handbook] if you're interested in block development. It is surprisingly well written.

Blocks are written either primarily in PHP or JavaScript (with js being the modern standard). The development process deviates greatly from the classic WordPress development paradigm. It utilizes Node.js, ESNext (cutting-edge JavaScript features), React (JSX), and webpack (code compilation).

File Structure

Blocks are almost always added in plugins. A typical project contains a src folder, which includes raw, uncompiled code and other assets. This folder does not have to be included in the final plugin. The contents of the src directory are used in the build process, which generates the final files in a separate build directory. The build directory contains the compiled files used to display the block. Here's the structure presented visually:



Source: [File structure of a block]

The index.js file gets loaded in the block editor. It's responsible for registering the block on the client side and typically imports the edit.js and save.js files to get the functions required for block registration.

The edit.js file contains the React component responsible for rendering the block in the block editor. The save.js file exports the function that returns the static HTML markup that gets saved in the post_content column of the database.

The style.css file contains the styles of the block that will be loaded in both the block editor and on the frontend. The editor.css will be loaded only in the editor. .scss and .sass files can be used instead.

render.php is used for dynamic blocks, which are rendered on the server using PHP instead of being built with JavaScript.

The view.js file will be loaded on the frontend when the block is displayed. You can put any js functionality your block needs in this file.

The names of those files can be changed. The files themselves are specified in block.json or as arguments when registering the block.

Attributes

Block attributes work just like HTML attributes. They allow blocks to be populated with user-provided content or to modify the block's settings or behavior in any way imaginable. Take a look at the example block ub/click-to-tweet mentioned in the Blocks section above. The attributes are 'blockID', 'ubTweet', and 'padding'.

Attributes are usually stored in a JSON format inside the block's delimiter, just like you would add attributes to an HTML tag. These attributes are ones that have been modified by the user. Default attributes don't need to be present in the block declaration.

Some attributes can be stored in the content of the block. This is the way the core paragraph block stores the text attribute (the paragraph block was shown in the Blocks section). You can see that it doesn't need a JSON in its delimiter. The attribute is stored in the generated HTML. This requires additional configuration when defining the attribute in the block.json file, namely the "source" and "selector" properties.

Attributes are defined by the block's author. When WordPress parses the content and renders the block on the frontend, these attributes are provided as parameters to the block's code. The block can then change its rendering based on the attributes, like displaying a string attribute inside a <p> tag or changing the padding of a given element. The block's author has full control of what attributes they define and how they use them in their block.

block.json

This is arguably the most important file for a block. It defines metadata about the block and the files associated with it.

Some of the most important metadata defined in this file are:

- **API version**
- **name**
- **title**

- **category**
- **attributes**
- **supports** (declaring support for certain native features, e.g., changing the text or bg color)
- and many more

block.json also allows you to specify files essential for the block's functionality. WordPress will then enqueue these files in the correct contexts. The names in brackets are the "standard" names of these files mentioned previously:

- **editorScript** - js file(s) loaded in the block editor (index.js).
- **editorStyle** - css file(s) loaded in the block editor (editor.css).
- **script** - js file(s) loaded both in the editor and on the frontend.
- **style** - css file(s) loaded both in the editor and on the frontend (style.css).
- **viewScript** - js file(s) loaded on the frontend (view.js).
- **render** - php file for a dynamic block (render.php).

These properties work either with a file path prefixed with "file:" or with a handle registered using wp_register_script() or wp_register_style() (more information on handles is available in the section on loading assets).

Registering The Block

Blocks in WordPress should be registered on both the server and the client side.

Server-side registration means registering the block in PHP. This process is pretty much the same as registering custom post types. You basically tell WordPress: "Here's a block type, it has this name and these parameters, remember it so that you can process it". WordPress then holds the information that your block type exists, along with its configuration, in memory, which allows it to do different things with that knowledge on the backend.

The registration itself usually takes place in the plugin's main PHP file. The function used to register a block is register_block_type($block_type, $args).

The $block_type argument is a string path to the directory containing your block.json file. This should be the path to the build directory, not the src directory (the block.json file should be copied to the build directory during compilation).

The $args argument is an array of optional parameters. These parameters are the same as in block.json. You can register your block fully with $args, without using the block.json file, but you shouldn't. It's considered legacy. Use the JSON file. In practice, the registration would typically look like this:

```
function register_my_block() {
      register_block_type( __DIR__ . '/build' );
```

```
}
add_action( 'init', 'register_my_block' );
```

Client-side registration means registering the block in JavaScript in the block editor. The concept of "registering" a block is a WordPress-specific thing. It means you create a js object with certain properties needed for your block to work in the block editor. You then pass this object to the editor's code. Remember, this happens only in the admin panel whenever the block editor is displayed. It doesn't happen on the user-facing frontend.

The registration happens in the editorScript file (typically index.js). To do it, you use the registerBlockType(blockNameOrMetadata, settings) method from the @wordpress/blocks JavaScript package.

The blockNameOrMetadata argument is either a string or an object. If it's a string, it's the name of the block you specified in block.json (e.g., 'ub/click-to-tweet'). If that's the case, and the block has been registered on the server, then the metadata will be automatically loaded from the metadata registered on the backend. Instead of passing a string, you can also pass a block.json object. You can literally import your block.json file to the index.js file and pass it as the first argument. The first approach is the best standard.

The settings argument is an object. It has many parameters. As a matter of fact, you could register your block without registering it on the server and passing only the name to the blocknameOrMetadata argument. That's because the settings argument allows you to specify all of the same metadata you specify in the block.json file. It's the same as with PHP registration. You could also pass the metadata in the $args argument there, but you shouldn't.

The settings argument has 2 extremely important properties:

- **edit** - the React component that gets used in the editor for the block. It's responsible for making the block function in the block editor.
- **save** - the function that returns the static HTML of the block to be saved in the database (for static blocks).

That's the code you write in the edit.js and save.js files. You have to import those files into index.js, and then pass the edit React component and the save function in the settings object. A typical index.js file before compilation might look something like this:

```
import { registerBlockType } from '@wordpress/blocks';
import './style.scss';
import Edit from './edit';
import save from './save';
import metadata from './block.json';
```

```
registerBlockType( metadata.name, {
      edit: Edit,
      save,
} );
```

You can theoretically register a block on the client side only. While you could do that, you shouldn't, unless you know what you're doing and have a very specific reason to do so. Some of the disadvantages that come from not registering the block on the server are:

- The block doesn't appear in the Block Type REST API endpoint.
- The block has to be static (no server-side rendering).
- The standard best way for registering the block in index.js doesn't work (you have to pass the block.json file instead of just the name).
- Block hooks don't work.
- Global styles (from theme.json in block themes) and aren't applied.
- and probably more…

Static Blocks

A static block doesn't require any rendering on the server (in PHP). The HTML of the block is generated in the block editor by the save() function (supplied in the index.js file when registering the block on the client side). Static blocks are rendered fully with JavaScript (in the save function), and their final HTML is stored in the post_content column in the database. Their contents are served directly from the database when rendering the page on the frontend - no additional computation is required. One example of a static block is the paragraph block:

```
<!-- wp:paragraph -->
<p>Welcome to WordPress. This is your first post. Edit or delete it, then
start writing!</p>
<!-- /wp:paragraph -->
```

When WordPress parses the post's content to render the final HTML, all it does is strip away the block delimiter (the HTML comments), so that only the <p> tag is rendered (the comments are never output on the frontend).

Static blocks don't have any ability to change after their markup has been rendered in the editor. Only truly static pieces of content that don't require server-side logic can therefore be static blocks. That being said, static blocks are more performant, precisely because they don't require server-side computation. You should try to make your blocks static whenever you can.

Block Validation

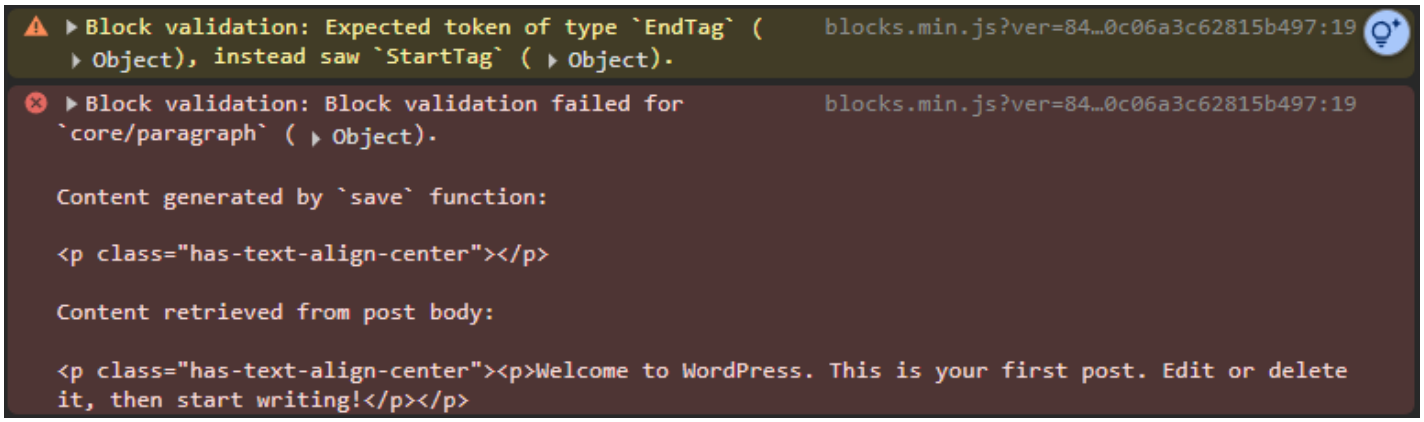
Block validation is a process the block editor uses to check the validity of static blocks. The existing blocks' save() function is run every time the editor is loaded. Remember, this function is

responsible for generating the block's HTML from the attributes supplied. If the returned HTML is different than the HTML stored in the database, the block is marked as invalid and the editor displays this:

Let's break the paragraph block to see how it works. We'll set the alignment attribute of the paragraph to center, which centers the text. This is what the block looks like in code:

```
<!-- wp:paragraph {"align":"center"} -->
<p class="has-text-align-center">Welcome to WordPress. This is your first post. Edit or delete it, then start writing!</p>
<!-- /wp:paragraph -->
```

As you can see, the "align: center" property has been added in the block delimiter and the has-text-align-center class has been added to the rendered markup. Let's now manually delete 'class="has-text-align-center"' from the <p> tag while keeping the align attribute in the delimiter and let's refresh the editor. Sure enough, we get a validation error. WordPress prints validation errors to the console so let's take a look:



As you can see, the markup generated by the save() function did not match the markup stored in the database (which, for some reason, has double <p> tags - likely a quirk of the editor).

You now have 3 options:

- **Attempt recovery** - WordPress will attempt to recover the correct markup of the block (it's not always successful). We'll do that in a second.
- **Convert to HTML** - converts the markup to a plain HTML block.

- **Convert to Classic Block** - converts the block to a Classic Block. This block is basically the Classic Editor in the form of a block. It uses the TinyMCE editor just like the classic editor did.

Here's what we get if we attempt recovery:

```
<!-- wp:paragraph {"align":"center"} -->
<p class="has-text-align-center"></p>
<!-- /wp:paragraph -->
```

An empty centered paragraph block… Underwhelming, but you can't really blame WordPress. If you take a look at what the save() function returned (shown on the screenshot from the console above), you will see that it's exactly what it should be. An empty <p> tag with the "has-text-align-center" class.

As a side note for the curious, the <p> tag was empty because the outside <p> tag of the invalid block stored in the database had no text. It had a <p> tag as its child which had text, but the parent <p> did not have any text itself, so the content attribute for the paragraph block was empty (the attribute is retrieved from the markup using something like document.querySelector('p').textContent).

So why do validation errors happen? Well, one possibility may be what we've done. The user plays with the raw code making it invalid. Another one is a flaw in the block's code resulting in the re-generated HTML not being the same as the stored HTML. But there's a more problematic reason - the block's author changed its save() function. If the save() function's output changes, for example after a plugin's update, the resulting markup generated might be different, which would mark the stored markup as invalid. What do you do then?

Block deprecation is a technique for updating blocks without making them invalid. The magic depends on creating a new deprecation object (usually in a deprecation.js file). This object needs to contain the old, deprecated save() function. You then register an array of such objects in the 'settings' parameter to the registerBlockType method. You can even change the names of your attributes using this method.

Whenever an invalid block is detected in the editor, all the deprecated save() functions are called. If one of them produces HTML matching the one stored in the database, the invalid content warning is not displayed. Instead, the markup of the block is automagically replaced with the new, updated markup when the content is saved (by clicking "Save" in the editor). This means the user has to save the content of the post for the block to upgrade itself to the new version.

It's worth adding that block validation is the single most important reason why many developers are reluctant to write static blocks and opt for dynamic blocks instead.

Dynamic Blocks

Dynamic blocks are rendered on the server each time the page is requested. Their HTML is not stored in the database. The block displayed in the editor is still written in javascript (in the edit.js file), but the markup returned to the frontend is generated in PHP. Note that this means you have to maintain 2 files, both responsible for displaying the block, but in different contexts. The save() js function should return null (although it can return HTML, which would make the static content act as a fallback, but that's an advanced topic).

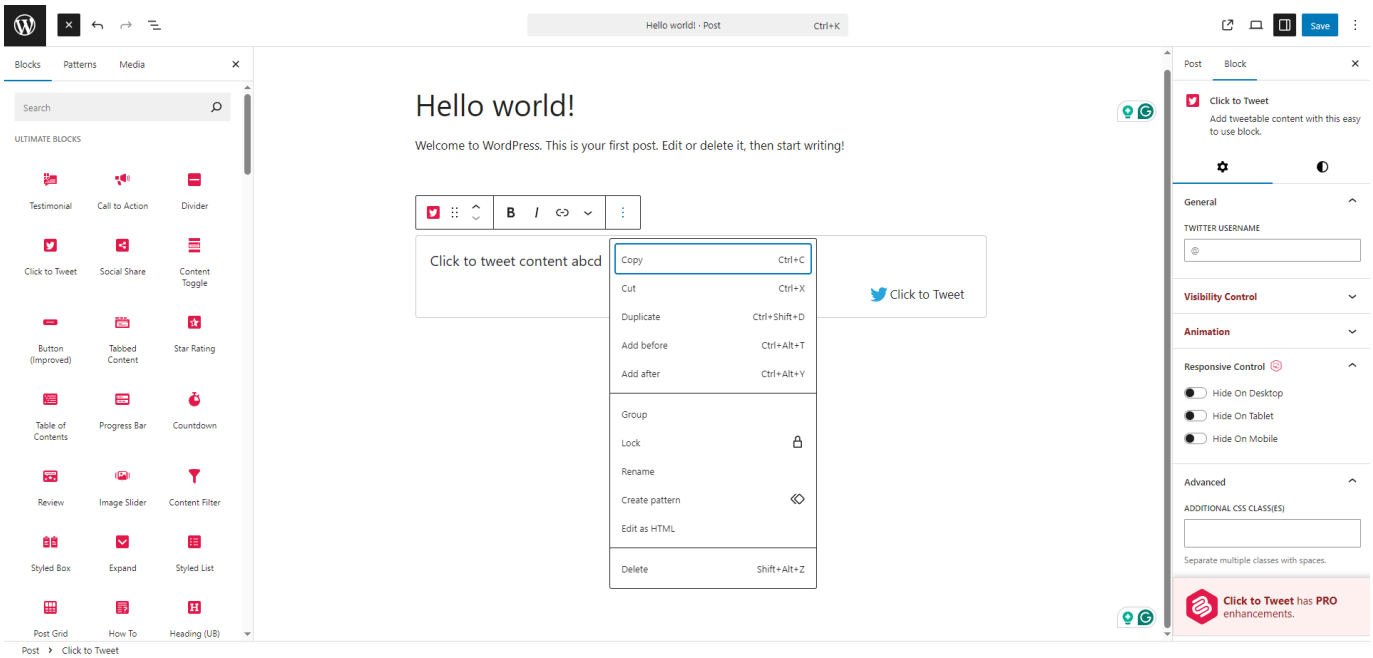
The PHP code to be run is specified either using the legacy 'render_callback' method when registering the block with register_block_type or using the 'render' property in block.json. The file specified in the 'render' property gets included when WordPress parses the content and encounters the block. The file should just echo the block's markup.

An example of a dynamic block is the aforementioned ub/click-to-tweet:

```
<!-- wp:ub/click-to-tweet
{"blockID":"5545214f-81c0-4e12-8325-6f0c79da171c","ubTweet":"Content to
tweet","padding":{"top":"0","bottom":"0"}} /-->
```

The Block Editor

You should already have a pretty good grasp of how the block editor works if you understand blocks. This editor, in contrast to the classic editor, is not just a <textarea> element. It's an entire React application. Here's what it looks like:



This document is not a tutorial and I won't explain every single detail you see here. Go check out the official [WordPress Block Editor article] if that's what you're interested in.

The editor is divided into 4 primary sections:

- **The top toolbar** - allows you to open the left sidebar, go back to the admin panel, save the post, change your editor's settings, and more.
- **The left sidebar** - the Block Inserter (open in the screenshot) allows you to choose from blocks, patterns, and media. The Document Overview (the 3 horizontal lines icon in the top toolbar) lets you see the structure of your blocks in a list and the content (headings) structure of your document.
- **The content area** - this is the main area in the center. That's where you put your blocks and write your content.
- **The right sidebar (also known as the Inspector)** - the Post tab allows you to edit the post's settings, such as the featured image, status, slug, associated terms, etc. The Block tab (visible in the screenshot) lets you modify the attributes of the selected block.

You can modify the appearance and behavior of your editor by clicking the Options button in the top toolbar (the 3 dots icon). There, you can switch from the visual editor to the code editor, which lets you modify the markup of the blocks. Here's what that might look like:

If you compare the block editor to the classic editor, you can see that the difference is huge. And not only in UI, but the entire way of thinking about creating content. If you know anything about people, then you know they don't like changes. The WordPress team anticipated this, and they released the "Classic Editor" plugin, which quickly became one of the most popular plugins in WordPress history, with over 9 million active installations as of 2025.

You might sometimes hear the Block Editor being referred to as Gutenberg. This is a very common point of confusion and is worth clarifying. Gutenberg was the name for the block editor while it was in development. When Gutenberg got officially merged with WordPress in 2018, the name changed to just "The Block Editor". Now, Gutenberg is a separate plugin you can download. This plugin serves as the testing version of the Block Editor. New features are introduced into the plugin and when they are ripe enough, they get merged into the WordPress Block Editor.

Block Styles

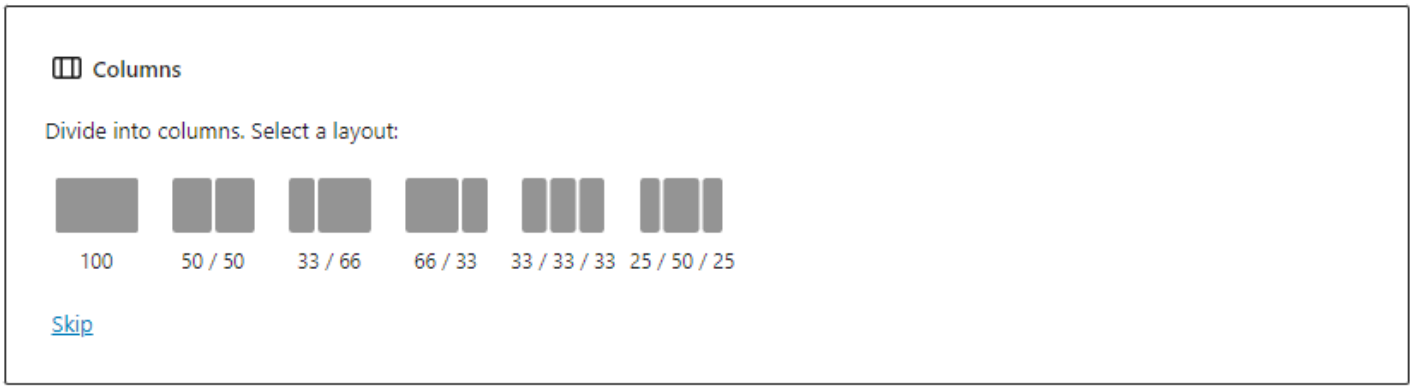
Plugin and theme authors can create custom styles for existing blocks. When a block has block styles registered, they appear as options in the right sidebar of the block editor. Block styles are registered either in PHP using register_block_style() or in JavaScript using registerBlockStyle(). Server-side registration is the standard. Let's register a "Rounded" style for the core button block:

```
function my_theme_register_block_styles() {
      register_block_style( 'core/button', array(
            'name'         => 'rounded',
            'label'        => 'Rounded',
            'is_default'   => false
      ) );
}
add_action( 'init', 'my_theme_register_block_styles' );
```

The effect of this function is that the block's wrapper will have an 'is-style-{name}' class added to it, so for us it'll be 'is-style-rounded'. The style author is then responsible for styling this class and loading the CSS in the editor and on the frontend. This can be achieved in many ways, such as using style.css, theme.json, inline CSS, wp_enqueue_block_style(), and more.

Block Variations

Block variations allow block developers to provide pre-configured versions of a block. The perfect example is the Columns block. When you insert this block, you get to choose a variation such as: 100, 50/50, 33/33/33, etc. Block variations are not their own blocks. They are a set of attributes to be automatically applied to the block the variation is created for. Variations are registered in the editor using the registerBlockVariation() function.



Block Patterns & Synced Patterns

Patterns are pre-defined components structured from blocks. There are 2 types of patterns: block patterns and synced patterns (known as reusable components before WordPress 6.3).

A Block Pattern is a pattern you can inject into your content. It works like a dumb copy-paste. Once you inject the pattern, there is no proof it was a pattern. If you change the content or structure of one instance of this pattern, no other instances are affected.

Moreover, if you change the registered pattern itself, the already used implementations are not affected. It's as if you pasted a set of blocks. Block patterns are commonly used for ready-to-go designs, like a testimonial card or an "about us" section.

A Synced Pattern is the counterpart to the Block Pattern. All instances of such a pattern are synchronized with the pattern's definition. If you modify the default content or the structure of the pattern, all instances will automatically be updated.

Since WordPress 6.6, instances of Synced Patterns can have their own content, i.e., you can change the text or image displayed in a single instance of the pattern. This functionality is called "overrides" and it's based on the Block Bindings API, which lets you bind dynamic data to blocks' attributes (covered later).

Registering Patterns

There are 3 primary ways to register a pattern - in a PHP function, in a PHP file, and in the Block Editor.

To register a pattern in PHP, you have to call the register_block_pattern() function. This function accepts 2 parameters - the name of the pattern and an array of options. The only required options are 'title' and 'content'. The title is self-explanatory. Content is a string containing the HTML for the pattern (you could copy that from the code editor). Note that you can only register standard Block Patterns this way. You can't register Synced Patterns in code (at least not in any standard way).

Another way of registering a standard Pattern is by placing it in the /patterns folder inside your theme directory. This is mostly used for block themes, but it works in classic themes as well. WordPress scans all the files in this folder. A pattern needs to be a PHP file with correct header comments. The options looked for in the comments are almost the same as arguments accepted by the register_block_pattern() function, i.e. title, slug (name), categories, description, etc. You don't need 'content', as the entire PHP file is the content.

The last way of registering a pattern is in the Block Editor itself, without any code. You can do that by selecting the blocks you wish to use in the pattern, clicking the 3 dots in the block toolbar, and clicking "Create pattern". This will open a pop-up, allowing you to name the pattern and choose whether it should be synced or not. Patterns registered in the editor are stored in the database as posts with the wp_block post type.

Synced Pattern's markup looks more like dynamic blocks rather than static blocks. That's to say they don't contain any static markup. They only include a 'ref' attribute, which points to the ID (post_id) of that pattern. Here's what one might look like:

```
<!-- wp:block {"ref":65} /-->
```

Block Bindings API

How do you display some dynamic content with a static block? Like a custom field or the user's name? You can't. You have to develop a full dynamic block, just for this silly little functionality. Yet, your custom block will literally just be a <p> element with dynamic text. It's so frustrating to have to do that. The core paragraph block already exists! If only we could somehow connect this block with the dynamic data. That would be neat, wouldn't it? Well, we can…

The block bindings API, introduced in WordPress 6.5, is one of the most, if not *the* most, revolutionary features since the introduction of the Block Editor. It allows you to connect blocks with dynamic data endpoints called sources. With block bindings, you no longer have to create a dynamic block to display dynamic data. We've already mentioned this API when discussing pattern overrides.

Displaying Custom Fields

Let's understand how this API works by looking at the first core way of using it - displaying post metadata.

To get a good grasp of it, we need to start by explaining some key concepts. The API is supported only by a few core blocks. There are plans for expanding that support and allowing custom blocks to support it. Only specific attributes of those blocks can be bound to data sources.

Here's a list of blocks along with compatible attributes that supported the API when it first got released:

- **Paragraph** - content
- **Heading** - content
- **Image** - url, alt, title
- **Button** - url, text, linkTarget, rel

These may seem underwhelming, but you can still achieve a lot with just this solid baseline. Imagine using the image, heading, and paragraph blocks, to create a section displaying information about a book's author - all from the metadata.

A data source is just a callback function. This was originally a PHP function, but it can also be a JavaScript function since WordPress 6.7. This function is responsible for returning the data to be used in the attribute it is bound to. The actual binding happens by adding an attribute to the block's delimiter.

Alright, we covered the foundation, it's time to see how it really works. It makes the most sense to start with a code example and explain it later. Here's a paragraph block with its content attribute bound to the thm_author_name custom field (expanded for readability):

```
<!-- wp:paragraph {
      "metadata":{
            "bindings":{
                  "content":{
                        "source":"core/post-meta",
                        "args":{
                              "key":"thm_author_name"
                        }
                  }
            }
      }
} -->
<p>Fallback text</p>
<!-- /wp:paragraph -->
```

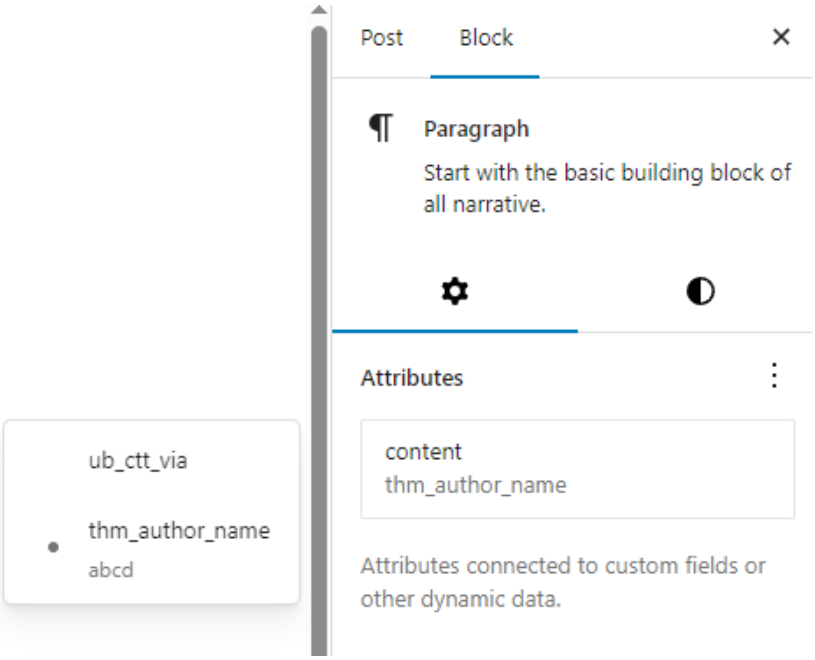
"content" is the attribute we're binding. For an image, it could be url, alt, or title. "source" is the registered name of the data source. In this case it's the core post meta. The "args" key houses an array of arguments that will be passed to the callback function. The callback's author gets to choose what arguments they are expecting. For core/post-meta, the most important argument is "key".

You can also see there's some fallback text inside the static <p>. This text will be rendered if the callback doesn't return a value. It's good practice to think about what your block should display if

the meta key doesn't exist. That's it. This paragraph will now display whatever meta_value is associated with the thm_author_name meta key.

It's important to note two things. First, a meta key has to be registered using the register_meta() function with args single => true and show_in_rest => true. Second, a meta key can't be hidden - that is, it can't start with an underscore. If one of those prerequisites is not met, the binding will not work.

It used to be the case that you could only create the binding using the code editor. This is no longer true, as WordPress provides a built-in "Attributes" field for blocks supporting block bindings, allowing you to create a binding in the visual editor.



Custom Sources

The number of available core sources will grow as the Block Bindings API matures. As of writing, it's in its infancy, so there is a big chance you will need a custom data source. Thankfully, that's not a problem. You can register your own sources and do whatever you like in their callbacks.

To register a custom source in PHP, you have to use the register_block_bindings_source() function. This function expects 2 parameters - $source_name and $source_properties. The name has to follow the namespace/name convention. $source_properties is an array of arguments.

The available arguments are:

- label

- get_value_callback
- uses_context

'label' is self-explanatory. 'get_value_callback' is the callback function for this source. We'll cover 'uses_context' later.

The callback's signature is:
function(array $source_args, WP_Block $block_instance, string $attribute_name)

- **$source_args** is the array of arguments specified in the metadata.bindings.$attribute.args. The core/post-meta would do $source_args['key'] to read the key.
- **$block_instance** is the WP_Block instance of the bound block for which the callback is being executed.
- **$attribute_name** is the name of the bound attribute. It would be "content" in our previous example.

Let's register a source to see how it works. We'll create a thm/date source, which is going to return the current date and time formatted according to our specification.

```
function thm_date_source_callback( $source_args ) {
	if ( isset( $source_args['format'] ) ) {
		return date( $source_args['format'] );
	}
}

function thm_register_source() {
	register_block_bindings_source(
		'thm/date',
		array(
			'label'              => 'Current date',
			'get_value_callback' => 'thm_date_source_callback',
		)
	);
}
add_action( 'init', 'thm_register_source' );
```

And here's what the paragraph block displaying that date looks like:

```
<!-- wp:paragraph {
	"metadata":{
		"bindings":{
			"content":{
```

```
                        "source":"thm/date",
                        "args":{
                              "format":"Y-m-d H:i:s"
                        }
                  }
            }
      }
} -->
<p></p>
<!-- /wp:paragraph -->
```

Well that was easy. Compare that to having to create a full custom dynamic block. Do you see why I said this API was the most significant update to the Block Editor so far? That being said, you can't edit custom sources in the visual editor (yet). I had to add these attributes manually in the block's markup using the code editor.

Context

The Block Context API is not specific to block bindings, but it doesn't really call for a top-level section of its own. It's a rather niche and advanced topic, but it's invaluable when presented with specific requirements. Don't worry if you don't fully understand it from this explanation alone.

When you're creating a template, the code in that template is executed in a certain context. Let's say we're viewing a single blog post. The code in that template operates in the context of that post. The title in this context is of that post. The ID is of that post. The content to be displayed is of that post. This is a very high-level context, encapsulating all of the code and elements on this page.

Now think smaller. Think about the individual blocks you place on your page. What if you wanted to display a list of related articles, and in that list you wanted to display the title and excerpt of other posts. It doesn't make sense to have to create new functionality for rendering the same data but for a different post. Instead, it makes more sense to change the context.

The Context API allows you to do that. It's based on the React Context API. A block can specify context it wants to pass to its children using the providesContext attribute in block.json. Similarly, children can specify contexts they are actively listening for using the usesContext attribute. A block like that will get access to these contexts if any of its parents, no matter how far away in the hierarchy, provides that context.

What does it look like in reality? Well, in the editScript JS file for the block, you can accept a context parameter in the edit function, and then use it like context['postId'], assuming the name of the context is postId. Similarly, WP_Block PHP objects have a context property, making it possible for you to use this context by doing $block->context['postId']. It's just an associative array of all contexts the block opted to accept and that have been passed by some parent.

A loop block can choose to provide the postId context. A context is usually linked to a block's attribute. It can then modify its postId attribute to something different, perhaps the ID of a related post it is currently displaying. All of the blocks inside of that loop listening for the postId context would then be able to use the new ID.

I'm avoiding directly referencing the Query Loop as I haven't introduced it yet, but if you know what that is - that's what I've been referring to the entire prior paragraph. Don't worry if you don't know what that is. You will soon.

But what the hell does that have to do with block binding? Well, remember the mysterious 'uses_context' argument available when registering custom data sources? That's it. It's an array of contexts' names. All of the contexts you specify here will be listened for by blocks which use this binding. This will then make the contexts available in the WP_Block $block_instance object passed as the second parameter to your callback.

Interactivity API

The Interactivity API, introduced in WordPress 6.5, is a JavaScript framework built into WordPress. Its goal is to provide a standard and seamless way of adding frontend interactivity to blocks, without relying on jQuery or custom js. It's very similar to Alpine.js, i.e., it's a declarative framework extending standard HTML with custom data attributes (directives). These attributes are used to store state and define actions taken on given events.

This is a rather large topic, and it's conceptually different from pretty much everything we've been discussing. Just imagine trying to learn a JavaScript framework in one chapter of a WordPress deep dive. Not going to happen. I'll cover it enough so that you know how it works, and if you want to learn it more intimately, I suggest you read the official Interactivity API documentation.

Reactivity & Directives

The foundation of this API is reactivity. You don't directly update the DOM. You declare the state the element depends on, and you declare actions taken on events that are responsible for changing that state. When the state changes, the elements react and update accordingly.

The most important abstract pieces of this system are:

- **State**
    - **Global state** - global data shared and accessible by any element.
    - **Local context (state)** - local data accessible only by the particular element and its children.
    - **Derived state** - data computed dynamically from the global or local state.
- **Actions** - functions, usually triggered by events, responsible for mutating state.

- **Bindings** - HTML elements are bound to state and updated automatically when the state changes.
- **Side effects** - optional callbacks executed when the state changes.

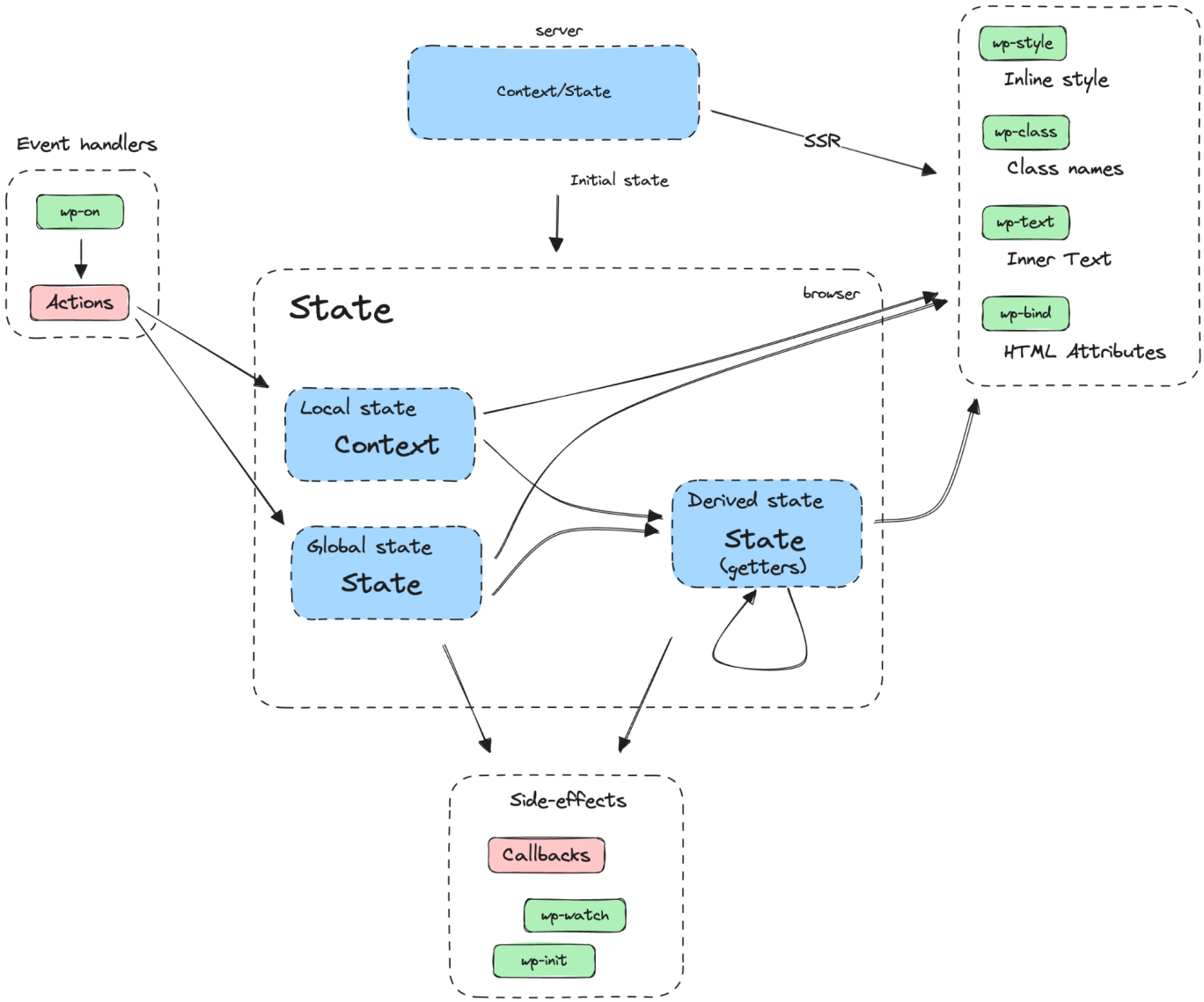
All of these parts are tied together using HTML data attributes, called directives. It's very important you understand what a directive is. Take this element:

```
<p data-wp-interactive="thm/myBlock">abc</p>
```

'data-wp-interactive' is a directive. These are attributes starting with data-wp. We'll talk about them without the data- prefix. Some of the most important attributes are:

- **wp-interactive** - enables interactivity for the element and specifies its namespace.
- **wp-context** - provides local context for the element.
- **wp-bind** - binds an HTML attribute with a state.
- **wp-class** - adds or removes a class based on a boolean value of a state.
- **wp-style** - adds or removes inline style based on the value of a state.
- **wp-text** - binds the inner text of the element with a state.
- **wp-on** - runs an action on a specified event.
- **wp-watch** - runs a callback when the state changes.

That's only the beginning of a long list, but it's all we need for a good understanding of this system. Consult the documentation to learn more. Look at this diagram of how the different parts of this system interact with each other. Maybe it will help you conceptualize it.



source: [Interactivity API Reference]

Code Example Using Local State

What better way to understand it than to actually use it, am I right? We'll create a simple dynamic counter block. This block will consist of only 2 elements: a button increasing the counter, and a span displaying it. I will not show you any of the code needed to register and create the block. Only the relevant parts - mostly the block's frontend HTML.

Here is this block's render.php file. We use PHP to render the block because it's a dynamic block:

```
<div
    data-wp-interactive="thm/counter"
```

```
    <?php
    echo get_block_wrapper_attributes();
    echo wp_interactivity_data_wp_context( array( 'counter' => 0 ) );
    ?>
>

    <button data-wp-on--click="actions.increment">Increment</button>
    <span data-wp-text="context.counter"></span>
</div>
```

Here is our code in the view.js file loaded on the frontend (you should load it using block.json):

```
import { store, getContext } from "@wordpress/interactivity";

store( 'thm/counter', {
    actions: {
        increment: () => {
            const context = getContext();
            context.counter += 1;
        },
    },
} );
```

And here is the final rendered HTML:

```
<div data-wp-interactive="thm/counter" data-wp-context='{ "counter": 0 }'>
<button data-wp-on--click="actions.increment">Increment</button>
<span data-wp-text="context.counter">0</span>
</div>
```

Okay, let's decipher what's going on here. First of all, the wp_interactivity_data_wp_context() function used in render.php is responsible for transforming an associative array into a formatted context directive. You could have just as well written data-wp-context yourself, but it's better practice to use this function (it handles JSON encoding and sanitization of the attributes). Our local state - the context, is defined and stored only in this wp-context directive. We only have one state property - 'counter', with a default value of 0.

The view.js file loaded on the frontend creates a store object for the thm/counter namespace. A store is the highest-level abstract object in this API. It defines the global state, actions, and callbacks. We create an increment action, which, as you can see, is just a callback function.

This function starts by getting the context of the element that called it using getContext(), and then increases the counter of this context.

The final HTML uses 4 directives. 'wp-interactive' activates the API and specifies the namespace. 'wp-context' defines the local state. 'wp-on--click' declares that the actions.increment action should be called on the 'click' event on the button. The naming convention is wp-on--{event}, so for other events you could do wp-on--input, wp-on--keyup, etc. Finally, 'wp-text' binds the 'counter' state to the span's inner text.

So, what will happen when we click the button? Here's the timeline:

1. The 'increment' action will be executed, modifying the counter.
2. The wp-context directive will be updated, with the 'counter' state being incremented by one.
3. The API will automagically update the span's inner text to the new value.

That's it. We have successfully created an interactive counter element without a single getElementById() call. Explaining why a declarative approach is better than an imperative one is beyond the scope of this guide. Go read a few articles on modern js frameworks if you're interested in learning more.

Code Example Using Global & Derived State

The previous example allowed us to place multiple counter blocks on the same page with each having its own counter. Let's modify it to use global state - making one button update all of the counters on the page. You could imagine it being like a button for time travel. You'd want all of your clocks updated, not just one.

Global state stores the data in the store instead of the inline context. The state should be initialized on the server. Here's our new render.php file:

```
wp_interactivity_state( 'thm/counter', array(
    'counter' => 0,
) );

<div
    data-wp-interactive="thm/counter"
    <?php echo get_block_wrapper_attributes(); ?>
>

    <button data-wp-on--click="actions.increment">Increment</button>
    <span data-wp-text="state.counter"></span>
</div>
```

Here is the view.js:

```
import { store } from "@wordpress/interactivity";

const { state } = store( 'thm/counter', {
    actions: {
        increment: () => {
            state.counter += 1;
        },
    },
} );
```

And here is the final rendered HTML:

```
<div data-wp-interactive="thm/counter">
<button data-wp-on--click="actions.increment">Increment</button>
<span data-wp-text="state.counter">0</span>
</div>
```

The wp_interactivity_state() adds state to the global store object with the thm/counter namespace. We got rid of the wp-context directive, as we are not using local context anymore. Similarly, context.counter was changed to state.counter, which is a reference to the global state.

By the way, if you've been scratching your head wondering: "what the hell is a namespace?" - let me clear that up for you. Every plugin/author/element should define a namespace for their store to avoid name collisions. That doesn't mean the store is only accessible to elements in this namespace. You can access another namespace's counter by doing namespace::state.counter (in a directive).

That's it. If we now placed multiple counter blocks on the same page, and incremented one of them, all of their counters would be incremented. That's because they all share the same global state by referencing the 'counter' variable in the thm/counter store.

Let's add more functionality to our block. How about we add a new span displaying double the value of the counter. You might be tempted to create a new state variable called counterDouble, but that would be wrong. If we did that, we'd have to remember to update the state twice in the action. We're opening ourselves up for state synchronizations issues. Instead, we can use derived state.

Here is our updated view.js code:

```
import { store } from "@wordpress/interactivity";

const { state } = store( 'thm/counter', {
    state: {
        get double() {
            return state.counter * 2;
        },
    },
    actions: {
        increment: () => {
            state.counter += 1;
        },
    },
} );
```

Notice the new state object and a double() getter function defined inside. That's our derived state. It uses an already existing state to dynamically derive the return value. Here's how our render.php changes:

```
wp_interactivity_state( 'thm/counter', array(
    'counter' => 0,
    'double' => 0 * 2,
) );

<div
    data-wp-interactive="thm/counter"
    <?php echo get_block_wrapper_attributes(); ?>
>

    <button data-wp-on--click="actions.increment">Increment</button>
    <span data-wp-text="state.counter"></span>
    <span data-wp-text="state.double"></span>
</div>
```

And here is the final rendered HTML:

```
<div data-wp-interactive="thm/counter">
<button data-wp-on--click="actions.increment">Increment</button>
<span data-wp-text="state.counter">0</span>
<span data-wp-text="state.double">0</span>
```

```
</div>
```

You can see that we use state.double the exact same way as state.counter. As a matter of fact, there is absolutely no difference between these 2 types of state from the outside point of view. You should use derived state whenever the data depends solely on data from other state variables.

You can see we nonetheless initialize the 'double' in wp_interactivity_state(). That's just the initial value, and you should usually define it - either statically or by computing it in PHP if you need to. How the value will later be computed on the frontend is defined solely in the state.double() function.

The API is smart. It knows that state.double is dependent on state.counter. Every time state.counter mutates, state.double will do too. That means our span displaying the double will automatically update every time we increment the counter.

Callbacks

Let's add just one more thing to our block. We'll make it log the counter's value in the console every time it is incremented. For that, we'll use a callback. Here's our updated view.js (shortened):

```
import { store } from "@wordpress/interactivity";

const { state } = store( 'thm/counter', {
    // [...] state and actions
    callbacks: {
        logCounter: () => {
            console.log('Current counter: ' + state.counter);
        },
    },
} );
```

We then attach this callback with the wp-watch directive to the element with the wp-interactive directive. I will not insult your intelligence by showing you the entire render.php file again. Here is the rendered HTML:

```
<div data-wp-interactive="thm/counter" data-wp-watch="callbacks.logCounter">
<button data-wp-on--click="actions.increment">Increment</button>
<span data-wp-text="state.counter">0</span>
<span data-wp-text="state.double">0</span>
</div>
```

That's it. The callback will be executed when the element is first created, and then each time state.counter changes. Again, the API is smart. The callback will only be called when the state it depends on mutates (the state used inside it, in our case it's only state.conter). You can use callbacks with local context as well.

Server Side Rendering

You might be wondering "Why?". Why create a proprietary JavaScript framework when there are so many available options? The answer lies in the requirements of this API. WordPress is still deeply rooted in PHP. Hooks need to have a way of modifying the output for plugins to be possible. None of the existing frameworks are PHP-friendly or compatible with WordPress.

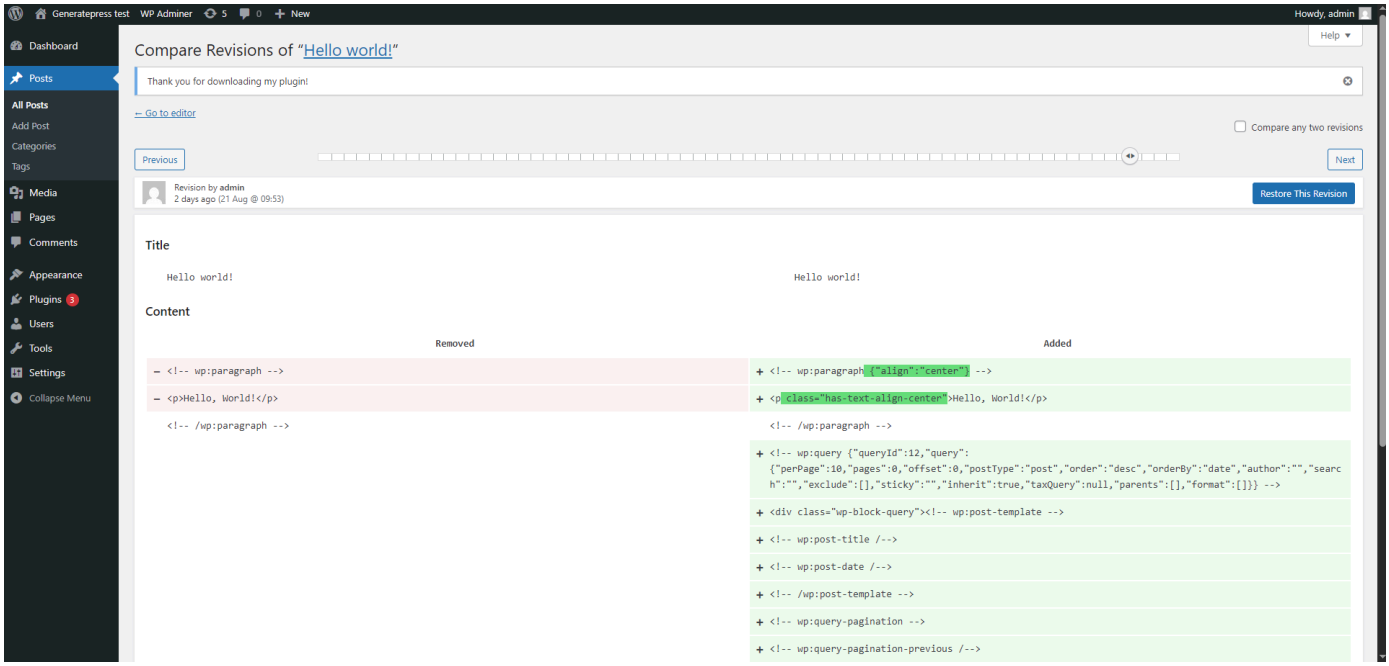
There's a reason why the "final rendered HTML" snippets in the examples above already contained the "0" in the span elements. The 0 is already there when the HTML is sent to the browser. That's right - the directives are parsed and rendered on the backend using PHP. The API starts its job on the server, and continues working on the frontend. Other frameworks usually require a JavaScript-based server, such as NodeJS, to achieve the same thing.

But the WordPress team is smart. They didn't reinvent the wheel. At the very foundation of this API sits Preact - a fast, minimalist react alternative. Overall, the API runs on about ~10 KB of JavaScript. And that JavaScript is shared across all blocks. If all block authors switched to using the Interactivity API instead of enqueuing react, svelte, or other libraries - that 10 KB is all we would ever need to easily make our blocks interactive.

PS: This API is not limited to blocks. It's just JavaScript code and HTML data attributes. You absolutely can use it in non-block HTML, but it was written with blocks in mind, since blocks are the future of WordPress.

Post Revisions

Revisions are WordPress's version control system for posts. With revisions, you can see a comparison between the markup of two different versions and choose to go back to an earlier version. Here's what that looks like:



This post has 70 revisions, which is why there are so many rectangles on the slider at the top. Using this slider, you can preview any of those revisions. You can even compare any two arbitrary revisions by checking the "Compare any two revisions" option. The diffs are visually marked with red and green. You can restore the version on the right by clicking "Restore This Revision".

It's important to understand how revisions work. The wp_posts table contains 3 crucial columns: post_status, post_parent, and post_type.

Every revision has a **post_type** of 'revision'. The **post_status** is equal to 'publish' on the published post, but the status of revisions is 'inherit'. This means they inherit the status of their parent, which makes for a smooth transition to the last relevant column. The **post_parent** column stores the ID of the main post (the one with the status 'publish').

Let's go through this system step by step to better understand it:

1. You click "Add Post".
    a. A new entry is added in the wp_posts table with post_status 'draft'. Let's assume the ID of this entry is 10.
2. You click "Publish".
    a. The post_status for the post with ID 10 gets changed to 'publish'.
    b. A new entry is created with ID 11. Its content is exactly the same as the one with ID 10. Its post_type is 'revision', post_status 'inherit', and post_parent '10'.
3. You change the contents of your post and click "Update".
    a. The entry with ID 10 gets updated.
    b. A new revision with ID 12 is created. Its content is equal to the new content of the post with ID 10.

Allow me to explain it once again, as this system might feel a little counter-intuitive. With revisions, there are always 2 posts in the database with the current content - the main published post (ID 10) and the latest revision. The revision is not created for the old version when you update. It's created for the new version. This revision, containing the new version, becomes the old version when you update the post again.

The main post with the status 'publish' never changes its ID. It's always the one created when you first added the post. Revisions are additive. Every new revision is a new post. This is also the case if you restore an old revision. When you do that, it's as if you pasted the contents of the revision into the editor. The main post's content changes, a new revision containing the same content is created, and the previous revision becomes the one with the content from before you clicked "Restore".

Most built-in post types support revisions natively:

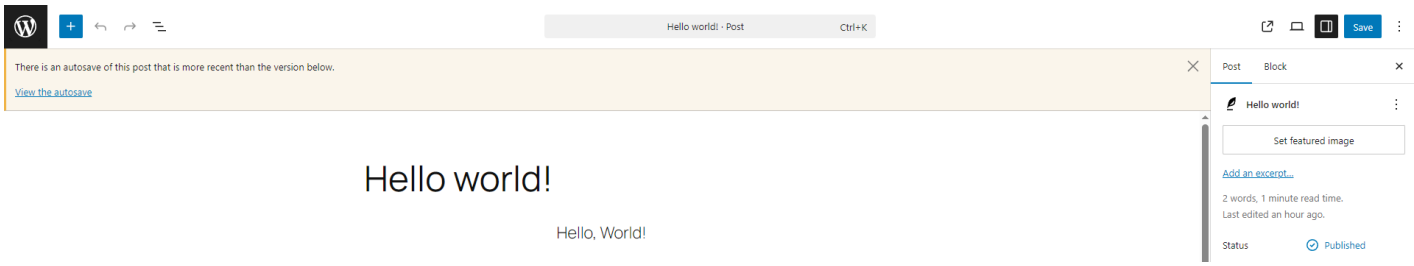
- **Posts**
- **Pages**
- **Block patterns**
- **Block navigation menus** (part of block themes)
- **Block templates** (part of block themes)
- **Block template parts** (part of block themes)
- **Global styles** (part of block themes)

Interestingly enough, custom post types do *not* support revisions by default. If you want your CPTs to have revisions, you have to specify it in the 'supports' parameter, like so:
'supports' => array(…, 'revisions', …),

Autosaves

Autosaves are an interesting part of this system. The WordPress Editor takes an autosave of your content every 60 seconds. This is by default, according to my testing, enabled only for posts and pages. An autosave is just a revision.

There is always at most one autosave in the database. The new autosave overrides the old one. The reason this feature exists is obviously to prevent loss of content in the unfortunate event of exiting/crashing without saving. If an autosave for the post exists, WordPress will let you know the next time you open the editor.



Custom Fields Revisions

WordPress 6.4 added native support for revisions of post metadata. When registering your meta (using either register_meta() or register_post_meta()), you can specify a 'revisions_enabled' argument. If you set it to true (it's false by default), your metadata will be revisioned.

The revision works similarly to post revisions. A new entry is added in the wp_postmeta table. The meta_key is the same - thm_author_name for our custom field. The difference is the post_id, which points to the revision post instead of the main post.

The revision is added only with a post revision, i.e., if you change only the custom field without changing the post, no revision will be stored. That's precisely because the revision of the metadata is connected with the revision of the post. It doesn't exist on its own, just like a post revision can't exist without a main post.

WordPress doesn't check if the meta_value has changed or not. It stores a meta revision for every post revision. Note that if you have many custom fields for a post type, and all of them have revisions enabled, it can quickly swell up the size of your wp_postmeta table. Every post revision will add n new rows, where n is the number of custom fields using revisions.

There is no UI preview for the revisioned metadata, even in the revision panel (when choosing a revision to restore). You have to either check the database manually to see what values are stored for the given revision, or just restore it and check the fields.

Advanced Revisions Management

You can modify how the revisions system works. In your wp-config.php, you can define the 'WP_POST_REVISIONS' constant. Let's look at a few options:

- define('WP_POST_REVISIONS', true) - enables revisions (default).
- define('WP_POST_REVISIONS', false) - disables revisions completely.
- define('WP_POST_REVISIONS', 10) - sets a maximum number of revisions to 10 (WordPress will delete the oldest ones beyond this number).

You can also change the autosave interval using the 'AUTOSAVE_INTERVAL' constant, like so: define('AUTOSAVE_INTERVAL', 120). This modifies it from the default 60 seconds to 120 seconds.

Themes

Themes are the presentation layer of a WordPress website. A well-coded theme should be completely replaceable. That is, you should be able to activate a different theme compatible with your content and still have a functioning website. No critical functionality should be dependent on a theme. Critical functionality should be added with plugins.

The previous paragraph describes the golden grail of WordPress development. The reality is messier. Ready-to-go mega-themes sold on sites like ThemeForest sacrifice this ideal for simplicity and speed of configuration. They bring along many problems, one of them being "theme lock-in". Your website becomes dependent on the specific theme.

Other themes act as separate creation frameworks inside WordPress. The most obvious examples are page builders, like bricks builder or divi. They also introduce theme lock-in, but typically allow for more customization and can be good options in certain circumstances.

This section is concerned purely with what themes are and how they work. The canonical and up-to-date info on themes is available in the official WordPress Theme Handbook.

style.css

style.css is the first of 2 files required for a theme to even show up in the Themes tab of the admin panel. Contrary to its name, its primary purpose is not to define styles. It's to define metadata about the theme. WordPress parses this file's comments looking for things such as theme name, version, author, etc. This file needs to exist, but doesn't need to have any content for the theme to work. Example structure:

```
/*
Theme Name: Twenty Twenty
Theme URI: https://wordpress.org/themes/twentytwenty/
Author: the WordPress team
Author URI: https://wordpress.org/
Description: Our default theme for 2020 is designed to take full advantage of...
Tags: blog, one-column, custom-background, custom-colors, custom-logo,
custom-menu, editor-style
Version: 1.3
Requires at least: 5.0
Tested up to: 5.4
Requires PHP: 7.0
License: GNU General Public License v2 or later
License URI: http://www.gnu.org/licenses/gpl-2.0.html
Text Domain: twentytwenty
*/
```

Keep in mind, style.css is not automatically loaded on the frontend. If you want to use it to write CSS, you have to explicitly enqueue it. More information, along with descriptions of each property, is available in the official WordPress style.css documentation.

index.php (or index.html)

index.php (or index.html for a block theme) is the second of 2 files required for the theme to show up in the Themes tab. It's the fallback template for every template hierarchy. You should make it work like a generic posts archive. index.html needs to be placed in the /templates/ directory for a block theme to work.

functions.php

functions.php is an optional, but extremely useful theme file. It's loaded in wp-settings.php when loading the WordPress core into memory. You can put any php code in there, but you probably shouldn't. Remember - presentation only.

Some of the most common use cases of the file are:

- Enqueuing scripts and styles.
- Registering features supported by the theme.
- Theme-specific customizations with hooks and filters.
- Defining custom helper functions/classes.

Theme Support

add_theme_support($feature, $args) is an important function that allows your theme to specify support for certain opt-in WordPress features. $feature is a pre-defined name of the feature and $args is an array with options required for some features.

The list of all features is long, but here are a couple of the most important ones (full list of features is available in the add_theme_support() documentation):

- align-wide (enables wide and full-width alignment for blocks)
- title-tag (enables plugins and themes to manage the <title> tag)
- post-thumbnails (enables featured images)
- html5
- automatic-feed-links
- post-formats
- custom-background
- custom-header
- custom-logo

What this function does is it adds the feature to a global $_wp_theme_features array. The feature name becomes the key and the $args array becomes the value. You should not interact

with this array directly. To check theme support, the current_theme_supports() function is used. The core WordPress code is full of conditional checks with this function, i.e., if (current_theme_support('post-thumbnails')).

Loading Assets

You load static assets in WordPress by using the enqueue system, not by hard-coding HTML tags. It's designed to make the process more robust, namely:

- It allows for specifying dependencies and automatically loads them in the correct order.
- It loads a given asset only once, even if you enqueue it multiple times.

The system is based on handles. Every file you enqueue has to have a handle. It's a unique identifier for that file. WordPress will then check if it already has this handle in the queue, and if it has, it won't load it again.

Beware not to load the same file twice under different handles. WordPress does not check the URL of the file, only the handle. Look out for loading different files under the same handle as well. WordPress will only load the first file enqueued with a given handle. That's why you should make sure the handles of your theme's files are unique by using the theme slug, like 'theme-slug-handle'.

Read more about including assets in the official [WordPress Including Assets Documentation].

Loading Styles

You enqueue styles by calling this function:

```
wp_enqueue_style($handle, $src = '', $deps = array(), $ver = false, $media = 'all');
```

It registers the file with URL $src, dependencies $deps, and version $ver as having a handle $handle ($media is for specifying the <link>'s HTML 'media' attribute, e.g., screen, print, etc.). This is basically a shorthand for registering the file with wp_register_style() first and then enqueuing it with wp_enqueue_style($handle).

You never call this function directly. Instead, you call it inside another wrapper function, and then register this function with one of these hooks:

- **wp_enqueue_scripts** - for loading assets on the frontend. Beware the confusing name, it's for both scripts and styles.
- **admin_enqueue_scripts** - for loading assets in the admin panel (/wp-admin/).
- **login_enqueue_scripts** - for loading assets on the login page (wp-login.php).
- **enqueue_block_editor_assets** - for loading assets only within the block editor's iframe.
- **enqueue_block_assets** - for loading assets on the block editor screen and the frontend.
- And other more specialized ones.

This ensures the file is added to the queue when the code responsible for this system is already in memory. The files' HTML tags will then be printed in the correct place on the rendered page (in the <head>).

The final code snippet for loading a css/main.css file might look like this (you would put it inside functions.php):

```
function my_theme_assets() {
    wp_enqueue_style(
        'my-theme-style',                       // The handle
        get_theme_file_uri( 'css/main.css' ),  // The URL
        array( 'example-dependency-handle' ),  // Dependencies
        filemtime( get_theme_file_path( 'css/main.css' ) ) // Auto versioning trick
    );
}
add_action( 'wp_enqueue_scripts', 'my_theme_assets' );
```

You can also load inline CSS with wp_add_inline_style().

Loading Block Stylesheets

You can enqueue CSS files for specific blocks. To do that, you have to call the wp_enqueue_block_style() function. This function expects the block's name and an array of arguments (almost the same as arguments to wp_enqueue_style()).

The CSS will then be included on the frontend and in the editor whenever the block is used. You should only use this function as a last resort - when styling the block with theme.json is impossible or unmanageable. Read the official Block Stylesheets documentation if you need this functionality.

Loading Fonts

The easiest way to load fonts is using a font delivery service like Google Fonts. You just enqueue the URL they give you and you're done. But you shouldn't do it. Please self-host your fonts. It's faster, more robust, and most importantly, it's GDPR compliant (Google Fonts is not).

Fonts are loaded in CSS files, so the process for enqueueing them is exactly the same as for styles. You just need to create the file first if you're self-hosting. There are tools for that, like google-webfonts-helper.

Loading Scripts

The idea for loading JavaScript files is the same as for CSS files, but you use wp_enqueue_script() instead. With scripts, you also have the choice to load them either in the <head> or the <footer> of the page. To load inline js, you can use wp_add_inline_script().

Passing Data From PHP to JS

Sometimes your JavaScript needs some data that is available only on the backend. You could create an API endpoint to serve this data, but it's like using a sledgehammer to crack a nut.

The official and correct way to include data available in PHP for use in JavaScript is by using the wp_add_inline_script() function. You would write your js in php and "attach" it before an enqueued script (the one you need the data in). Keep in mind the timing of execution. If you enqueue the script in the head, but the data you need is not available at that point, then you won't be able to get it. In that case, you'd have to enqueue the script in the footer.

There's also an older way, which you will probably see way more often. It's using wp_localize_script(). This function takes a php array as an argument and renders an inline script with the equivalent JavaScript object or array. The original purpose of this function was exactly what it's named - localization (translations). The modern best practice is to load data using wp_add_inline_script(). This helps keep the code more semantically correct, which makes it more readable and self-documenting.

Loading Assets Conditionally

Unfortunately, including assets conditionally in WordPress kind of sucks. The way you do it is different depending on the context. There are 2 ways.

Template-level logic is when you need to include an asset for a specific template or page. You'd use WordPress Conditional Tags for that. Let's say you have a CSS file that is only used in your single.php template. You would enclose your wp_enqueue_style call in an if (is_single()) conditional tag. But what if you only need a style for a certain page? That's where the sucking happens. You would be forced to check either the slug or the ID of the page, both of which are brittle as they can change and are decoupled from the code of the actual page.

Component-level logic is when you need to include an asset for a specific component. This is way nicer as there are no conditional checks involved. The wp_enqueue_script() is called in the code of the component itself, which means it's automatically included on every page the component is used on. The component's code is run in the contents of the page, which means it's called after the <head> has already been rendered. ~~This limits the use of this technique to scripts loaded in the footer only, as styles can only be included in the head~~.

Update: My tests have shown that WordPress renders every asset left in the queue in wp_footer(). That means if you enqueue any script or style after wp_head() has run (e.g., in your php templates), it will be rendered in the <body>. This is confusing and quirky behavior, and can lead to unplanned script executions (if you set $in_footer = false thinking it won't be rendered) or invalid HTML (if you enqueue a style, which will render a <link> in the <body>).

The way you'd typically use component-level logic is by calling wp_enqueue_script() in your shortcode's code or the render_callback function of a dynamic block (if you implement your component as a block).

Getting File URLs

As you might've noticed in the code snippet above, you shouldn't hard-code your assets' URLs. If you do, good luck with domain name migrations. You should use one of these 3 functions:

- **get_stylesheet_uri()** - returns the active theme's style.css URL (child theme's style.css if it's active).
- **get_theme_file_uri($file)** - returns the active theme's URL, with an optional $file parameter. Falls back to the parent theme if a child theme is active and the file doesn't exist.
- **get_parent_theme_file_uri($file)** - same as the previous one, but this one searches the parent directory only.

The get_theme_file_uri() and get_parent_theme_file_uri() functions were introduced in WordPress 4.7. The way to include assets before that was by using either get_template_directory_uri() or get_stylesheet_directory_uri(). These functions return the URL to the parent or the active theme directory. That means you had to use string concatenation to get the file URL.

Most importantly, the get_stylesheet_directory_uri() does **not** fall back to the parent directory if the file doesn't exist in the child directory (like get_theme_file_uri() does). That means you can't easily overwrite an asset if the parent theme uses these functions. They are legacy functions. Do not use them (unless you know what you're doing).

Scripts Bundled With WordPress

WordPress bundles many custom and third-party scripts (e.g., jQuery). You should always use those bundled scripts to avoid loading the same script twice, thus creating conflicts with plugins. The full list of bundled scripts is available in the /wp-includes/script-loader.php file.

Classic Themes

Classic themes have been in use since 2005, and as of 2025, they are still more widely used than block themes. They use PHP files as templates. Some of the most commonly created files are functions.php, header.php, footer.php, single.php, and of course, the required index.php and style.css. The templates usually use The Loop to display content available in the global $wp_query object.

The topic of how to write a good classic theme is outside the scope of this document. Remember, this isn't a tutorial.

The Loop

The Loop is one of the most famous and fundamental WordPress concepts. It's how you display the content from the database in each template. Let's cut to the chase.

The Loop is just an iterator pattern. It's not something you can directly touch. There's no the_loop() function. In this case, it's best to start with a code snippet:

```
if ( have_posts() ) :
    while ( have_posts() ) : the_post();
        the_content();
    endwhile;
else :
    echo 'No posts to display';
endif;
```

That's it. It's a while loop. The function have_posts() checks if there are any posts left to iterate over in the global $wp_query variable. The function the_post() sets the global $post variable to the next post in order. You display the contents of each post inside the while loop (in whatever way you want to display it).

By convention, the_post() is called on the same line as the while loop. Functionally, it's the same as calling it on the first line of the loop. You should also use the endwhile syntax instead of curly braces. The reason is simple - you're going to be putting HTML inside that loop, and it's way easier to find where the loop ends with an endwhile instead of a closing brace (the same rule applies to conditionals).

Let's now imagine we create an archive-thm_book.php file inside our theme. According to the template hierarchy, this file will be used to display the archive page for our custom book post type. The global $wp_query object will hold n posts of type book (up to 10 by default). We can now create the HTML of the "book card" inside The Loop, just like product cards in e-commerce stores. By iterating over this HTML in The Loop, we'll display a list of books on our books archive page. The user will be able to see it once they head over to the example.com/books/ page.

$post & Template Tags

WordPress defines an important global variable - $post. It's a variable of type WP_Post. It holds the currently iterated over post in The Loop, and the first post in $wp_query before The Loop is run. It is populated by the the_post() function during each iteration. You can interact with it directly, but you shouldn't. You should use template tags instead.

Template tags are just php functions meant to be used in templates to get dynamic data. Fancy name, simple idea. Some template tags are not based on $post, e.g., bloginfo($param),

get_header(), get_footer(), etc., but most are, e.g., the_title(), the_content(), get_the_author(), get_the_date(), etc. One of the most important template tags is the_content(), which renders the contents of the post's post_content column in the database.

Template tags prefixed with 'get_the_' return the requested data (usually as a string). You can store this data in a variable and do something with it before displaying it. Template tags prefixed with 'the_' display the data for you. They are, in essence, just wrappers echoing their 'get_the_' equivalents.

As already noted, before running The Loop, the global $post contains the first post returned by $wp_query. That is convenient for single posts (such as the single.php template), as you don't have to run The Loop. You can just use template tags directly, because $post is set to the first (and only) post in $wp_query. This behavior can seem counter-intuitive for templates such as archive.php, where the_title() would not return the title of the archive, but of its first post.

Secondary Loops & WP_Query

Sometimes you need a different Loop than the one created automatically by WordPress. Some potential cases with that requirement are:

- Showing 5 most recent news articles on an about us page.
- Displaying related products at the bottom of a product page.
- Creating a complex front page with different sections for products, featured posts, categories, etc.

You can't get that data from The Loop, as those posts aren't part of the query run by WordPress for the given page. In that case, you need to create a custom query. You can use the WP_Query class to achieve that.

To run a custom query, you need to create an object of this class and pass an $args array to its constructor. The query will be run, and the posts will be available in the object. You can fetch them and run a Loop the same way you do in The Loop - with have_posts() and the_post() (but called on the query object). As a matter of fact, the global $wp_query object is an object of this class, and the main query is created in the same way.

Code Example

In this case, it's best to start with a code example and explain it later.

```
<h3>Latest News Stories</h3>
<ul>

<?php
// Step 1: Define the arguments for our custom query.
$args = [
```

```
    'post_type'         => 'post',   // We want standard posts.
    'posts_per_page'    => 5,        // We want 5 of them.
    'category_name'     => 'news',   // Only from the 'news' category.
    'orderby'           => 'date',   // Order them by publication date.
    'order'             => 'DESC',   // The newest ones first.
];

// Step 2: Create a new WP_Query object.
$news_query = new WP_Query( $args );

// Step 3: Run a new Loop using our query's methods.
if ( $news_query->have_posts() ) :
    while ( $news_query->have_posts() ) :
        $news_query->the_post(); // This sets up the global $post.
        ?>

        <li>
            <a href="<?php the_permalink(); ?>">
                <?php the_title(); ?>
            </a>
        </li>

    <?php
    endwhile;
endif;

// Step 4: Restore the original post data.
wp_reset_postdata();
?>

</ul>
```

$args

This is the array of arguments that controls what the final SQL query looks like. It provides a lot of control. Some of the most commonly used options are:

- **post_type**
- **post_status**
- **name** (retrieves a post by its slug)
- **post__in** (retrieves only posts with provided IDs)
- **post__not_in** (excludes posts with provided IDs)
- **posts_per_page** (number of posts for pagination)

- **paged** (the page number - if posts_per_page is 10 and paged is 3, posts 21-30 will be returned)
- **orderby**
- **order** (DESC or ASC)
- **cat** (retrieves posts from a specific category by its ID)
- **category_name** (same as cat but by slug)
- **tax_query** (complex taxonomy query, described in detail later)
- **meta_key** (retrieves posts that have a specified meta key)
- **meta_value** (the value to match to the meta_key)
- **meta_compare** (comparison operator, e.g, =, !=, >, <, LIKE, IN, etc.)
- **meta_query** (complex meta query, described in detail later)

This list is nowhere near complete. Go read the official WP_Query Documentation if you need a list of all parameters along with instructions on how to use them.

tax_query

The tax_query parameter allows you to filter the query based on multiple taxonomies with different relations. tax_query is an array. This array can have a 'relation' parameter and other arrays. The actual filtering is done in the inner arrays. It's hard to explain, just look at the code example:

```
// [...]
'tax_query' => array(
      'relation' => 'AND',
      array(
            'taxonomy' => 'category',
            'field' => 'slug',
            'terms' => 'technology',
            'operator' => 'IN',
      ),
      array(
            'taxonomy' => 'tag',
            'field' => 'slug',
            'terms' => 'opinion',
            'operator' => 'NOT IN',
      ),
),
// [...]
```

This query will make it so that all the posts returned are in the technology category AND do not have the opinion tag. The 'relation' parameter controls the relation between the inner arrays. The inner array has 5 options:

- **taxonomy** - the name of the taxonomy being searched.

- **field** - what the term is selected by. Possible values are 'term_id', 'name', 'slug', or 'term_taxonomy_id'.
- **terms** - taxonomy term(s).
- **operator** - operator to test. Possible values are 'IN', 'NOT IN', 'AND', 'EXISTS', and 'NOT EXISTS'.
- **include_children** - whether or not to include children for hierarchical taxonomies.

You can also nest outer arrays in the inner arrays to create more complex queries. For example, you could have: "return posts that are in the technology category OR are in the business category AND have an AI tag". The logical expression would be: (technology || (business && AI)). I'm not going to include a code example as that would be too long. Go read the documentation.

meta_query

meta_query is like tax_query but for metadata instead. It queries the wp_postmeta table. The most typical use case is to filter by data added with custom fields. The structure is the same as with tax_query. You have outer and inner arrays. You can have OR or AND relationships. You can nest outer arrays inside inner arrays, creating more complex queries.

The parameters in the inner array are:

- **key** - meta_key in the database.
- **value** - meta_value in the database.
- **compare** - operator to test. Possible values are '=', '!=', '>', '>=', '<', '<=', 'LIKE', 'NOT LIKE', 'IN', 'NOT IN', 'BETWEEN', 'NOT BETWEEN', 'EXISTS', 'NOT EXISTS', 'REGEXP', 'NOT REGEXP', and 'RLIKE'.
- **type** - the value type. Possible values are 'NUMERIC', 'BINARY', 'CHAR', 'DATE', 'DATETIME', 'DECIMAL', 'SIGNED', 'TIME', 'UNSIGNED'. This parameter is important as if you were to sort numbers with type 'CHAR', you'd get: 1, 2, 3, 32, 4, 49, 5, etc.

Running the Loop & wp_reset_postdata()

Mind the lowercase 't' in 'the Loop'. The name 'The Loop' is by convention the main loop based on the global $wp_query. Here we're talking about a secondary loop based on our custom query.

To run the main Loop, we could've just used the have_posts() and the_post() functions. These functions are nothing more than wrappers over $wp_query->have_posts() and $wp_query->the_post(). Running a Loop based on a different query is as simple as calling those methods on the other query's object, which is what we're doing here.

It's important to note that $news_query->the_post() sets the global $post variable to the current post, just like The Loop does. This is good, because it means you can use Template Tags. The side effect is that when your loop ends, the global $post holds the last post from your secondary Loop. If you were in the single.php template, and were to now call the_content() outside of any

loops, the content of the 'rogue' post would be rendered instead of the content of the main post the user wants to see. This problem is solved by the wp_reset_postdata() function.

wp_reset_postdata() resets the global $post variable to the one from the main $wp_query. If you run The Loop before your secondary Loop, then after you call this function, the $post will be the last post from the main query. If you call it before the main Loop, or you don't have The Loop at all (like in single.php), the $post will be restored to the first post in $wp_query. Bottom line - the $post goes back to whatever it was before you ran your custom loop. The rule is very simple. Always call wp_reset_postdata() after executing a secondary Loop. Not doing so can cause many very confusing bugs.

get_posts()

The WP_Query class is a powerful tool, but with great power comes great complexity. There's a much simpler function that suffices in many situations: get_posts($args). This function is just a wrapper over WP_Query. It is supposed to be used with just a few parameters, and it returns an array of posts (theoretically you could supply any parameter you can to WP_Query).

The only arguments advertised by the documentation are:

- **numberposts** - number of posts to be returned (alias of posts_per_page in WP_Query).
- **category** - category or categories specified by ID (alias of cat in WP_Query).
- **include** - array of post IDs to retrieve (alias of post__in in WP_Query).
- **exclude** - array of post IDs to exclude (alias of post__not_in in WP_Query).
- **suppress_filters** - if true, the query will *not* be passed through filters, such as pre_get_posts, posts_where, etc. Be careful, this argument is set to true by default (WP_Query also accepts suppress_filters, but it defaults to false there).

As already noted, this function returns an array of WP_Post objects (or post IDs if 'fields'=>'ids'). I'm not going to tell you what to do with that data. You should know what you're trying to achieve. But I'm going to show you how you can run a secondary Loop with that:

```
$my_posts = get_posts( $args );
if ( $my_posts ) {
    foreach ( $my_posts as $post ) { // Use a standard foreach loop
        setup_postdata( $post ); // Manually set up post data
        // ... do stuff with template tags ...
    }
}
wp_reset_postdata(); // Still need to reset!
```

This is the same thing as running a Loop directly on a WP_Query object. You can see we use setup_postdata() instead of the_post(). This is a more low-level function. It's actually used by WP_Query::the_post() to set the global $post variable. This means you still have to call wp_reset_postdata() after running such a loop.

If you want to run a simple loop where you don't need to use template tags, and you don't want to modify the global $post, you can do that as well. In that case just don't call setup_postdata(). You can then access post properties directly, such as: $post->ID, $post->post_title, etc.

Templates & Template Parts

As noted in the beginning of this guide, its purpose is not to teach you how to create WordPress websites, including coding themes. It would, however, be incomplete had I not cover templates and template parts at least to some extent. This section provides a glimpse into what writing a classic theme looks like. Consult the Theme Basics documentation for more pragmatic information (or just read the source code of some high-quality themes).

The most important thing to revisit is that themes are based on templates. Which template is used is predicated on the template hierarchy. You can opt to create as many specific templates as you like (single.php, page.php, category.php, etc.) or as little as just the index.php file.

Templates are usually a combination of PHP, HTML, and template tags. Template tags, as already explained, are functions used to display dynamic data in templates. This data doesn't have to come from the database (like it does with the_title() or the_content()). Another form dynamic data can take is PHP files, called template parts.

Template parts are an extremely important part of writing templates. You wouldn't want to copy-paste your header's and footer's HTML into every template you create, would you? That's what template parts are for. There are a couple of template parts with specific built-in template tags to retrieve them:

- **header.php** included by *get_header()*
- **footer.php** included by *get_footer()*
- **sidebar.php** included by *get_sidebar()*
- **searchform.php** included by *get_search_form()*

To include custom template parts, you should use the generic get_template_part($slug, $name, $args) function. To include the file parts/content.php (where the parts folder is located in your theme's directory), you would do:

```
get_template_part( 'parts/content' );
```

This allows you to keep your code DRY if you had to use the content.php file in multiple templates. The $name argument lets you specify a suffix added to the slug after a hyphen. If you were to pass 'page' as $name, the searched for file would be parts/content-page.php. This is very powerful if you want to let your users control which template parts gets used, such as:

```
// some code setting the $layout variable, perhaps based on a setting from
```

```
the Customizer (more on that later)

get_template_part( 'parts/content', $layout );
```

This way you could dynamically include either a content-wide.php or a content-narrow.php template part. One powerful feature of the get_template_part() function is that if the $name is specified, but the file is not found, it falls back to the file with just the slug. This may be used, for example, for including a specialized template part based on the post type: get_template_part('parts/content', get_post_type()). It will search for content-post.php, content-page.php, content-author.php, etc., and if it doesn't find one, it defaults to content.php.

The $args argument allows you to pass additional data to the template part. It's an array. If you were to pass an array with a key value pair 'key' => 'value', you could then do echo $args['key'] inside the template part to echo 'value'. It's worth noting the $args parameter is a relatively recent addition (WordPress 5.5). Before that, it was less straightforward to pass data to template parts. You might see older themes use tricks such as set_query_var() or global variables.

Page Templates

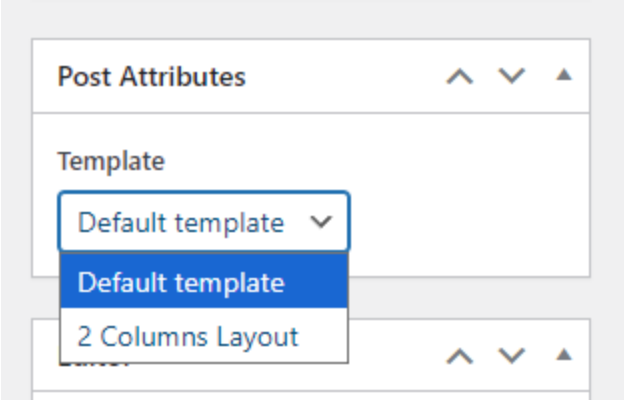
Custom templates (historically known as Page Templates) are template files meant to be chosen for a page by the user. This just means the user can select which page template the page should use and it'll be used. Look back at the Template Hierarchy. A custom template is at the very top of the hierarchy of a page.

This functionality was originally supported only by pages. Support for all post types was added in WordPress 4.7. You may still be confused about why this even exists. Let me give you a real world example: A theme wants to let the user choose a layout with either 2 or 3 columns. The theme author creates two templates: "2 Columns Layout" and "3 Columns Layout".

In reality, a Page Template is just any PHP file in the theme with a correct header comment. The file can be placed in a subfolder or not - WordPress will find it either way (a page-templates subfolder is a good practice). Here's what a custom_template.php file may look like:

```
<?php
/*
* Template Name: 2 Columns Layout
* Template Post Type: page, post
*/
```

That's it. The Template Post Type attribute is only supported since WP 4.7, and in this case, it makes the template available for pages and default posts. You can specify any post type, including custom ones. Here's what this looks like in the Classic Editor:



When the user selects the "2 Columns Layout" template and saves it, the custom_template.php file will be used for this page/post. Of course, it'll not display anything as this file contains only the comments, but you get the point.

Classic Widgets

Widgets provide a way to add modular content in specific, pre-defined areas of the theme. They are a big part of customizing classic themes. They are not relevant for block themes at all. There are two sides to widgets - widgets and sidebars.

Widgets are just blocks of PHP code. They are created by developers. They have to extend the WP_Widget class, implement methods such as widget(), form(), and update(), and then be registered using the register_widget() function. The widget() method, which is the only truly required method, is responsible for actually rendering the HTML of the widget on the frontend of the website. The form() and update() methods are responsible for giving the website's administrator ways of modifying the widget, such as changing the text or image rendered in the widget() method.

Sidebars have a very unfortunate name. They should've been named "Widget Areas" instead. Sidebars are places in the theme's code where widgets can be rendered. Think of it as shelves and books. You mount shelves on the walls in strictly defined spots. They are immovable. These are sidebars. Then you can place all sorts of different books on every shelf. The books are widgets.

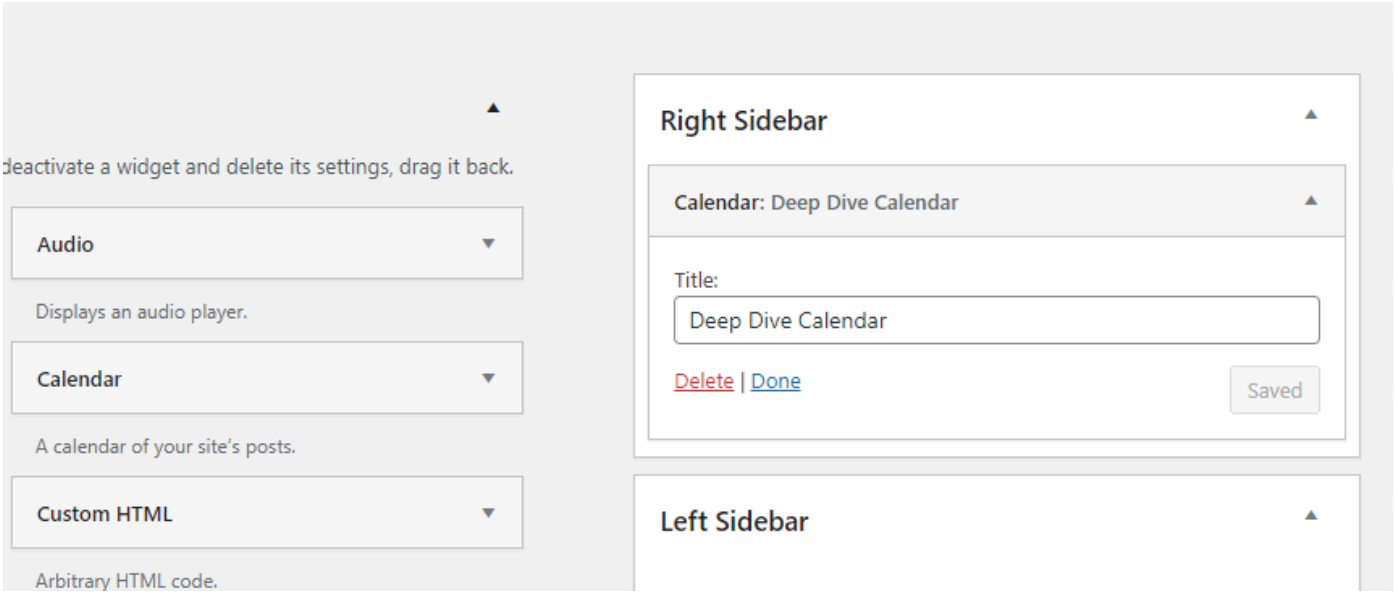
Sidebars do not have to be literal sidebars. That's why this name is so awful. You can define sidebars in the center of the page, or the footer, or anywhere else your soul might desire. It's the theme author's responsibility to define sidebars in places where the theme's users might want to place widgets.

Going into the code, defining sidebars has 2 parts. First, you have to define the existence of the sidebar. It's not like with hooks, where you could just call the hook out of the blue. You have to register the name of the sidebar first using register_sidebar(). This will make the sidebar appear in the Appearance -> Widgets menu.

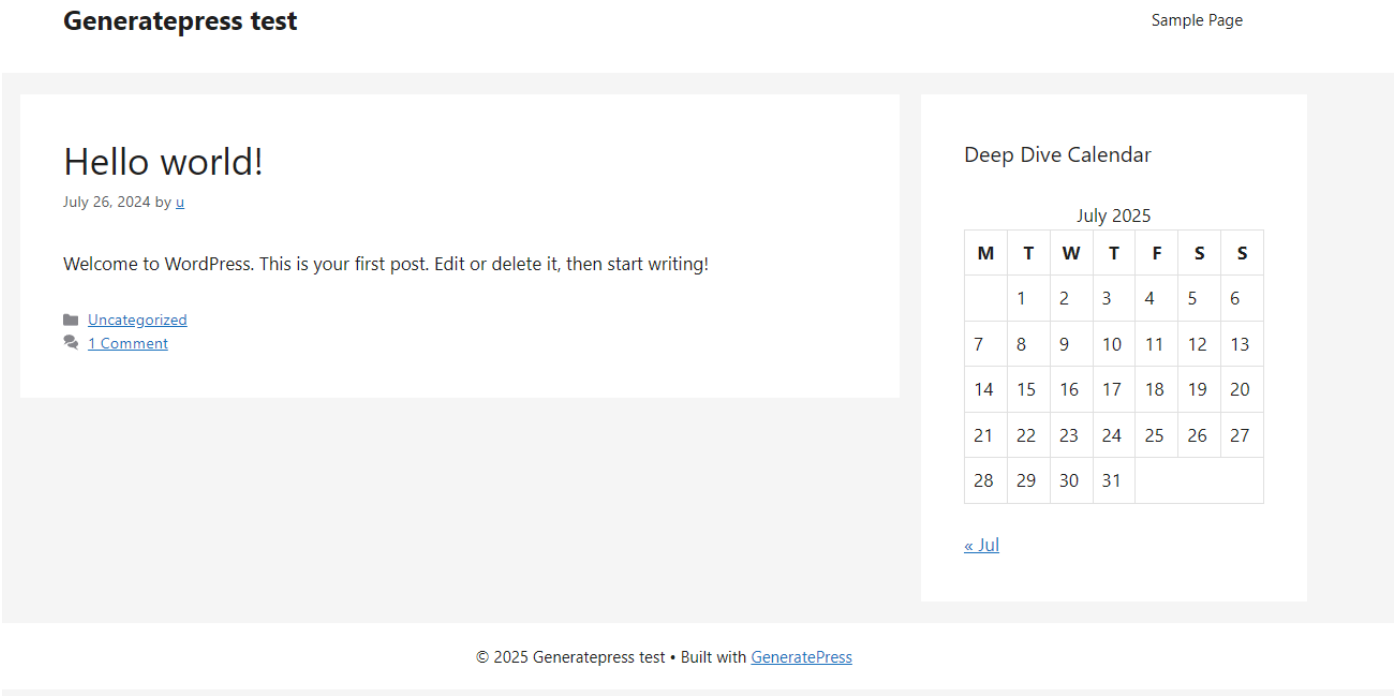
Only then can you render the widgets assigned to the sidebar in your template files using the dynamic_sidebar($sidebar) function. You would register the sidebar in your functions.php file, and then you would call dynamic_sidebar() in your theme's template files. Wherever you call this function, all the widgets assigned to the given sidebar by the website's admin would be rendered using their widget() method.

Let's look at the actual way of using widgets. The picture below shows a sliver of the classic Widgets menu in the admin panel. You can see widgets on the left (Audio, Calendar, and Custom HTML), and you can see sidebars on the right (Right Sidebar and Left Sidebar).

The widgets have been defined somewhere in the PHP code that is run on the website (either by WordPress core, a plugin, or a theme). The sidebars have been registered by the theme. To use a widget in a sidebar, you manually drag it over to the box of the sidebar you're interested in. Once there, you can see customization fields. For a Calendar widget, there's a 'Title' text field. That's defined by the widget's form() method.



Here's what the widget looks like on the frontend of the website. As you can see, it actually is a sidebar, because that's where the theme's authors called the dynamic_sidebar() function for the "Right Sidebar" area. The HTML of the widget is rendered by its widget() method. You can also see that the widget displays the title of my choice.

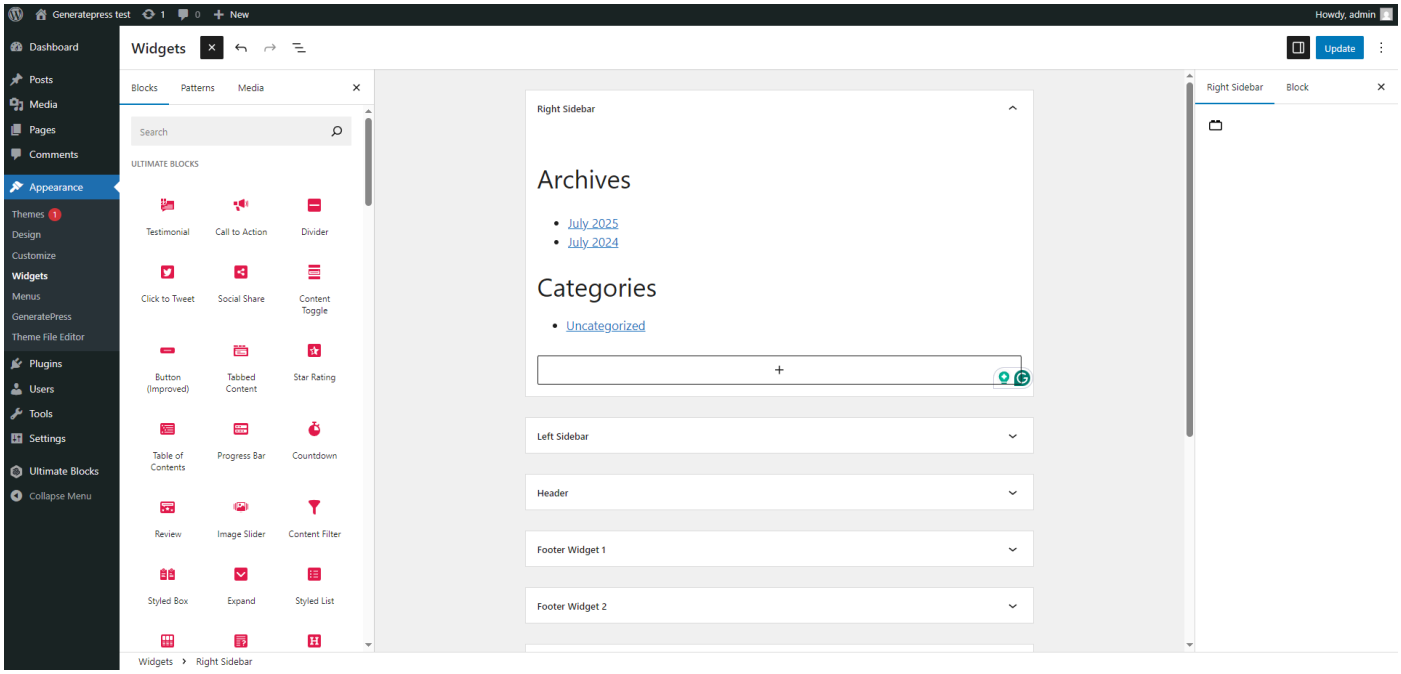


Widgets are a powerful and much used way of customizing classic themes. In practice, you'd rarely write a widget from scratch, unless you had a really good reason to. There are thousands of plugins whose whole job is to supply those widgets, like social media icons, newsletter signup forms, etc. Some themes come with custom widgets as well, but you have to be wary of using them, as that introduces theme lock-in. Remember, themes are for presentation only. WordPress core itself comes with quite a few default widgets out of the box (e.g., a search box, recent posts, custom HTML, etc.).

It's important to note that widgets are an exception to the "everything is a post" rule. They are not stored as posts. They are stored as configuration in the wp_options table.

Block Widgets

Block widgets have been introduced to unify the experience of creating content in the Block Editor with the experience of creating widgets. The biggest change is that they now utilize the Block Editor. Any block you can use in your posts, you can also use in your sidebars. Take a look at the new block-based widget page:

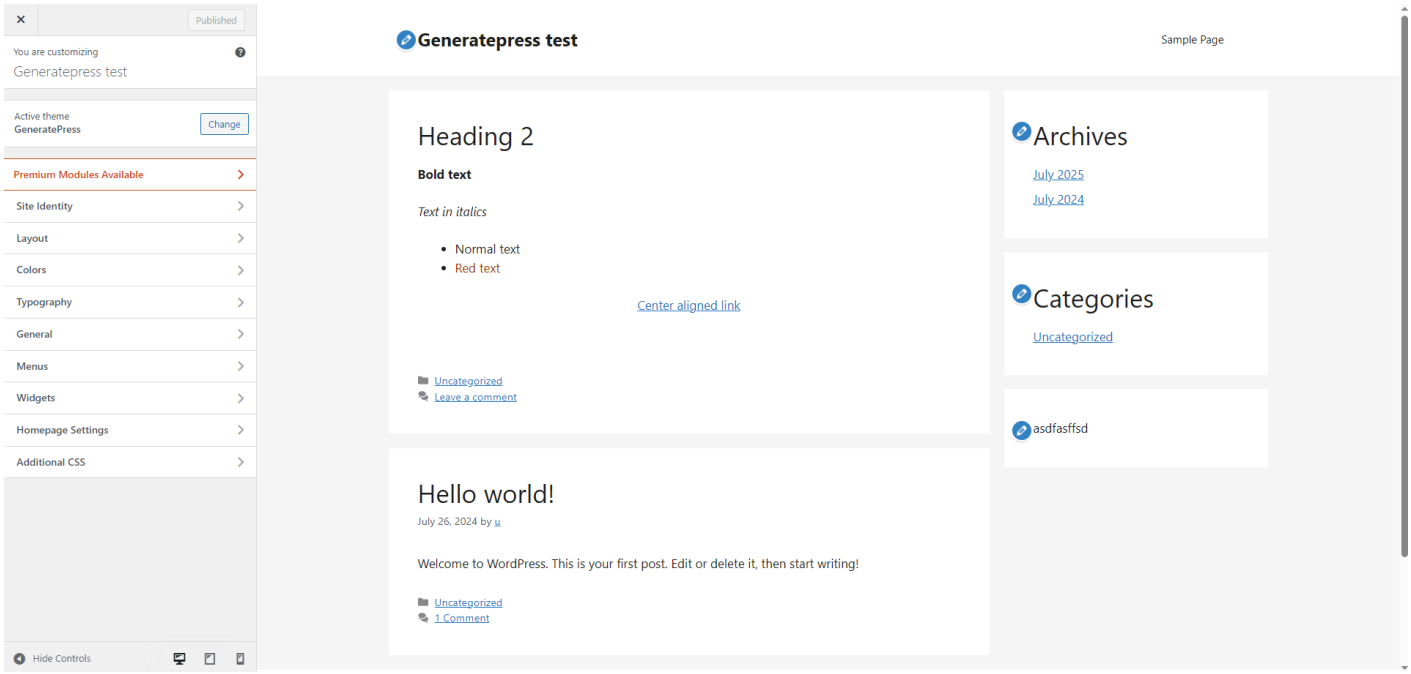


This change requires a shift in perspective when it comes to thinking about widgets. With classic widgets, you had to create the widget and then use the widget in your sidebars. Here you could say that every block is a widget. The word "widget" loses its meaning, and it's more useful to now think of sidebars as pre-defined spots not for widgets, but for more general content.

Block widgets are backward compatible with classic widgets. The classic widgets registered with PHP are wrapped with a special "Legacy Widget" block. This acts as the adapter between the block content and the old widgets. If you want to go back to the classic widgets view, you can use the official Classic Widgets plugin.

The Customizer

The Customizer allows you to customize options for classic themes. Its most powerful feature is live preview. Any change you make is displayed before you save it. That means you can change the logo, the colors, the layout, or the typography, and you will be able to preview the changes before publishing them. Options available for customization are defined by the theme author.



The Customizer is based on the Customize API. This is an object-oriented API you have to interact with in order to display options for users to change in your theme. In practice, that means adding an object to the WP_Customize_Manager object.

The Customizer is composed of 4 types of objects:

- Settings
- Controls
- Sections
- Panels

Settings

A setting is an object that represents a single piece of data you want to manage. It's the link between the user interface (the control) and the database. Think of it this way - if you want to let your users choose the number of posts displayed on a particular page, you have to register a setting - let's call it posts_displayed. That creates a WP_Customize_Setting object with this name as its ID.This object has all the necessary methods for saving the data in the database. Your job is only to register the setting. The WP_Customize_Manager takes care of calling all methods.

Settings have 4 primary responsibilities:

- Defining the data container and its default value.
- Handling data storage and retrieval.
- Sanitizing data.
- Handling live preview (more on that later).

When adding a setting, you're faced with the choice of its type. There are 2 setting types: 'theme_mod' and 'option'.

theme_mod is a setting specific to the current theme. If the user changes the theme, the setting becomes dormant (it doesn't get deleted but it can't be used in a different theme). theme_mod settings are stored in the wp_options table in the database as a serialized array of all settings for this specific theme.

option is a global setting supposed to be theme-agnostic. Settings of this type should rarely if ever be added by a theme. They are stored in the database as separate rows for every setting. These settings work exactly the same as normal WordPress options you can modify in Settings (site URL, blog name, etc.), except they are used in the Customizer interface. As a matter of fact, you can change settings such as the blog name in the Customizer. The benefit over the classic Settings menu is live preview.

Controls

A control is the UI element responsible for changing the setting in the Customizer. Every control object has to have an associated setting and an associated section. The most important argument when adding a control is its type. There are many different types, such as all types of HTML <input> elements, checkbox, textarea, radio, select, and dropdown-pages. You can also create your own controls by subclassing the WP_Customize_Control class. A type 'number' would be used for our posts_displayed setting. That would render <input type="number"> in the Customizer.

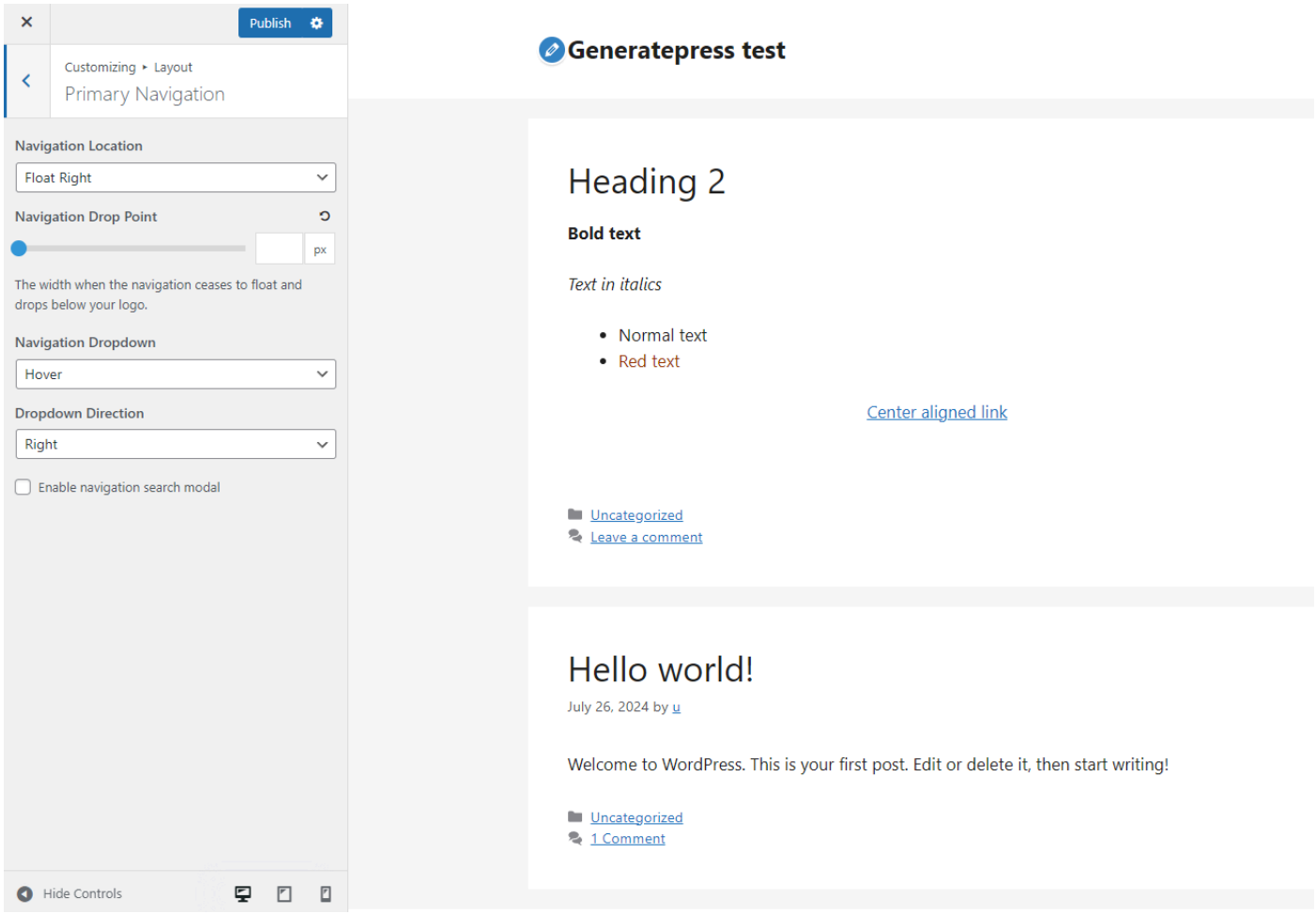
Sections

A section is just a UI container for controls. You can see many sections in the screenshot above: Site Identity, Colors, Typography, General, Homepage Settings, and Additional CSS. A section must contain at least 1 control to be displayed.

Panels

A panel is a UI container for sections. It's not required. If you find yourself thinking "I wish I could group sections" when designing your options, then you should use a panel. It's usually not needed. There are 3 panels in the screenshot above: Layout, Menus, and Widgets.

Originally, panels were supposed to be more than that. The documentation states: "More than simply grouping sections of controls, panels are designed to provide distinct contexts for the Customizer, such as Customizing Widgets, Menus, or perhaps in the future, editing posts". You can see that in practice by navigating to the Widgets panel. It doesn't display a simple list of options like sections do. It displays a list of all registered sidebars and allows you to modify the widgets inside. The full potential of panels (e.g., "editing posts") was never realised, perhaps because it got superseded by Full Site Editing with the Block Editor.



The screenshot above shows all of the ideas. You can see at the very top, we are in the Layout panel (written next to "Customizing"). We are in the Primary Navigation section. This section displays multiple controls with different types. These controls are connected to theme_mod settings that affect the appearance of the active theme, GeneratePress. All of that has been registered by the theme.

Live Preview

As already noted, a setting defines the live preview mode used to update it in the Customizer. This mode affects the preview's behavior. There are 3 options: refresh, postMessage, and selective refresh.

refresh is the default mode. If it is used, the entire preview <iframe> gets refreshed on every change to the setting. This means a new request to the server and re-rendering the entire page. It's easy for the developer as it doesn't require any additional code, but it provides a bad user experience for the person using the Customizer.

postMessage is the complete opposite. It's a JavaScript API allowing you to enqueue a script in the preview responsible for updating the element associated with the setting. There's no request to the server. The change is instantaneous. There are 2 major drawbacks. First, you have to write the JavaScript code to update the element. Second, it violates the DRY principle. You have to make the same updates as you do in your PHP template. This means you'll have to modify it if your template ever changes, and that's prone to errors.

Selective refresh is a hybrid method. It involves registering a callback function on the server responsible for returning the HTML markup just for the updated element. The Customizer requests this updated markup when the setting is modified and replaces the HTML in the preview with the new HTML. You have to specify the selector of the element to be updated when registering the callback, or more precisely - the partial, on the server.

Code Example

Let's register our posts_displayed option.

```
function thm_add_posts_displayed_customizer( $wp_customize ) {
  $wp_customize->add_setting( 'posts_displayed', array(
    'type' => 'theme_mod',
    'capability' => 'edit_theme_options',
    'default' => 10,
    'transport' => 'refresh',
    'sanitize_callback' => 'absint', // You should specify a sanitization callback
here
  ) );

  $wp_customize->add_control( 'posts_displayed', array(
    'type' => 'number',
    'section' => 'posts',
    'label' => 'Number of posts displayed',
    'active_callback' => 'thm_displays_posts', // If false, the control is not
rendered in the Customizer
  ) );

  $wp_customize->add_section( 'posts', array(
    'title' => 'Posts',
    'description' => 'Control appearance of posts-related content',
    'panel' => '', // Not typically needed.
    'capability' => 'edit_theme_options',
  ) );
}
add_action( 'customizer_register', 'thm_add_posts_displayed_customizer' );
```

Menus

Menus look simple, but the way they are implemented is surprising. Menus are a standard part of every website. They are the navigation links (usually placed in a <nav> element in the header and the footer). WordPress provides a way of creating a menu in the Appearance > Menus tab. A menu is just a list of links. You can add pages, posts, taxonomies, or even custom links. A theme provides menu locations, kind of like sidebars for widgets. The user then selects a registered place they want their menu to be rendered in. A <ul> element with all the links is then output.

To register a menu location, the theme has to start with calling the register_nav_menus($locations) function. This function takes an associative array as an argument, where the key is a programmatic name of the menu (e.g., header-menu) and the value is the corresponding human readable label (e.g., Header Menu). Upon registration, the menu location becomes available in the "Manage Locations" tab. Every location has a <select> element allowing the user to choose their menu.

To render the menu in a template, the theme has to call the wp_nav_menu($args) function. The $args array provides a lot of options. You can specify a user-created menu to be rendered, specify a container HTML element, CSS classes, and even additional text to be rendered before and after every element. The most important option for us is 'theme_location'. The value associated with that key is the menu location name you want to render (e.g., header-menu). The menu selected for that location by the user will then be displayed.

A code example could be:

```
// in functions.php
function thm_register_menus() {
	register_nav_menus( array(
		'header-menu' => 'Header Menu',
		'footer-menu'   => 'Footer Menu',
	) );
}
add_action( 'after_setup_theme', 'thm_register_menus' );

// somewhere in the header.php template
wp_nav_menu( array(
	'theme_location' => 'header-menu',
) );
```

Menus In The Database

Everything was simple so far, right? Well, brace yourself. The way menus are stored in the database is probably not what you think. A note before reading this section: you will probably

not find much practical use for a deep knowledge of how menus are stored., but it will build your understanding of the WordPress database and what really is possible with its default schema.

The first thing to understand is that menus use 6 tables in the database: wp_terms, wp_term_taxonomy, wp_term_relationships, wp_posts, wp_postmeta, and wp_options.

The menu itself is actually a term. It's stored in the wp_terms table with its name and slug. It belongs to a special taxonomy called nav_menu, and this connection is stored in the wp_term_taxonomy table. The wp_term_relationships table then links the menu with its items.

If you're attentive, you might be thinking "hmmm, but the wp_term_relationships table connects terms with posts". That is exactly right. Every item you add to a menu is a post. More precisely, it's a post of a special nav_menu_item post type. But the wp_posts table doesn't have nearly enough columns to cover all the data a menu item requires, such as the ID to the linked post or the custom URL. This is where wp_postmeta shines, storing all this additional information as metadata associated with the menu item.

Okay, so we have our menu stored in the database, but how is it connected to a menu location? Try to remember the way settings were stored in the Customizer. One of the ways of storing a setting was using theme_mod - an option in the wp_options table for the active theme. That is exactly how these assignments are stored. The theme_mod_{theme-slug} row in the wp_options table has a 'nav_menu_locations' entry (remember, the entire row is just a serialized array). This entry connects the names of the registered locations with the IDs of the menus stored in the wp_terms table.

Parent & Child Themes

This section is focused mostly on classic themes, as child themes are less useful for block themes. Nonetheless, the concepts described are universal to both types. Block theme-specific topics are discussed in the Block Themes section.

A theme acts as a parent theme when it has an active child theme. Any standalone theme can become a parent. We've been discussing standalone themes throughout this entire section. Let's assume we created a new theme called Awesome Theme and we started selling it. Users download it and activate it.

Now let's imagine a power user wanting to modify the look of the footer. They head to the wp-content/themes/awesome-theme folder and they modify the footer.php file. It works, they are happy. But then we push an update (version 1.1). They download it and POOF… All of their changes are gone. The parent theme files have been replaced with their new versions. That's what child themes are for.

Creating a Child Theme

It's surprisingly easy to create a child theme. There's just 1 file a child theme needs: style.css. index.php is not required (nor encouraged unless you want to modify the parent theme's index.php).

Let's create a child theme for our Awesome Theme. We go to wp-content/themes, and create a new directory: awesome-theme-child (the name doesn't matter). We create the style.css file inside this directory. There's only 1 additional header you have to add to style.css for the theme to be recognized as a child theme. It's the 'Template' header.

```
/*
Theme Name: Awesome Theme Child
Template: awesome-theme
// other style.css headers (as explained in the style.css section)
*/
```

The Template header has to match the directory name of the parent theme. WordPress basically makes this a child theme for the theme located at "wp-content/themes/$Template". The Theme Name and other options don't matter for the child theme to work.

How Child Themes Work

As already noted, the main purpose of a child theme is to provide a way for adding persistent changes to a theme. After you create the child theme directory with the style.css file, you have to head to Appearance > Themes and activate the child theme. That's right, our Awesome Theme should be inactive and the Awesome Theme Child should be active instead.

When WordPress is looking for templates (e.g., single.php, archive.php, etc.), it searches the child theme first, and if it doesn't find the file, it proceeds to look for it in the parent theme. The order in the hierarchy (child first, parent second) is what allows child themes to override their parents.

Outside of full templates, there are also other files that your theme users might want to modify. These are template parts, assets, JS files, etc. If you paid attention reading the section about loading assets, you might already know how that works.

The built-in functions for getting file paths and URLs are designed to search the child theme first. I'm not going to explain them again, but those are functions like get_theme_file_uri() and get_theme_file_path(). The same is true for get_template_part() used to include parts. That is the whole reason why we're using these functions instead of hard-coding the URLs or doing require('content.php').

Persistence of modifications is ensured by the fact that updates apply to the parent theme only. Child themes are not touched when an update is released and downloaded. The way you'd usually modify a file from the parent theme in the child theme is by copying the file to the child theme (with the same name and directory structure) and making your modifications.

Child themes are a double-edged sword. They are amazing if you only have to make a small amount of insignificant changes, but they become a nightmare if you're heavily modifying an actively maintained theme.

Just imagine you change 10 template parts. The parent theme is getting updated once a week. Every time you have to check if any of the modified parts have been updated. If they did, you have to analyze the changes and decide if you should merge them. Sometimes the updates will introduce breaking changes, which will make your files (which still contain the modified old versions) not compatible with the rest of the theme, and your site will break.

Trust me when I say this, as I have made that mistake - the less modifications in the child theme, the better. If you can add something with a widget, do it. If you can change an option in the Customizer, do it. If you can use a hook, do it. Try as hard as you can not to modify files directly in the child theme, and try to choose themes which will not force you to do so. If you find you need to heavily modify a theme, you might be better off forking the theme and maintaining it yourself.

functions.php In Child Themes

functions.php is a special case when it comes to files. It doesn't override the functions.php in the parent theme. Both of these files are loaded. The child's version is loaded first. This is powerful because it serves as your main PHP file for things like enqueueing assets or adding hooks - all while the parent's functions.php (which is usually a critical file) still runs.

The benefit of running the child's version first is that you can override so-called pluggable functions. Pluggable functions are functions enclosed in if (! function_exists('function_name')). Most themes will make all of their functions in functions.php pluggable, so that you can override them if you find you need to.

Some poorly coded themes will use get_stylesheet_uri() to enqueue their style.css file. If you remember this function from the section on loading assets, then you should know why it's a terrible mistake. As soon as a child theme is activated, this function will return the URL of the *child's* style.css. Suddenly, your website will have no styling, as the parent's style.css is not loaded. If you encounter that, you'll have to enqueue the parent's style.css yourself in your child's functions.php.

Block Themes

Block themes are themes built entirely out of blocks. They are the future of WordPress, but they have their own problems. To understand block themes, we have to first understand why they exist in the first place.

Blocks Themes vs Classic Themes

Think about what it's like to customize a classic theme. Want to change an option? You go to the Customizer. Want to create a menu? You go to Appearance > Menus. Want to add a piece of content in some place? You create a widget. Want to write a post? You write the post in the editor (potentially using shortcodes). Want to modify a template or a template part? Go write some PHP in a child theme. The experience is inconsistent and confusing. Block themes are supposed to solve that.

The underlying concept block themes are meant to enable is Full Site Editing (FSE). FSE is the idea that you can edit every part of your website in a consistent manner - no matter if it's a menu, page content, or a template. This is achieved by making everything a block. If all parts of your website are wrapped around the same abstraction (a block), you can treat them uniformly and provide a consistent experience.

Block Theme Structure

The structure of a block theme is a little different. The most important change is that templates have to be placed in the /templates folder, and template parts have to be placed in the /parts folder. There's also a new, very important file - theme.json. All the other rules still apply. You have to have a style.css file and you can create a functions.php file.

Templates & Template Parts

Templates is where the first major difference becomes apparent. Templates in classic themes were PHP files (.php). Templates in block themes are HTML files (.html). These files do not contain the final HTML markup. They contain block syntax. Remember the Code editor view of the Block Editor where you could see the block delimiters? That's what's in those files.

The template hierarchy is the same for both types of themes. The only difference is that WordPress searches for templates in the /templates directory instead of the root theme directory. WordPress knows that a theme is a block theme by checking if the /templates/index.html file exists. If it does, it's a block theme.

Template parts share the same characteristics as templates - they contain block markup and are searched for in the /parts directory. Parts aren't included using functions because there are no functions in HTML. Instead, there's a template part block:

```
<!-- wp:template-part {"slug":"header"} /-->
```

This block will get replaced with the contents of the /parts/header.html file. The slug attribute specifies the name of the file without the file extension.

If you remember the section about Synced Patterns, then you might think they are pretty similar to parts. You wouldn't be wrong. The only difference between the two is the fact that parts are primarily file-based, and Synced Patterns are stored only in the database.

There are, however, many internal talks about allowing Synced Patterns to be registered by themes (with files) instead of only by users. It's also discussed to merge the concepts of patterns and template parts into one. As a matter of fact, template parts are available under the "Patterns" menu in the Site Editor, and here's what the footer.html template part looks like in the Twenty Twenty Five theme:

```
<!-- wp:pattern {"slug":"twentytwentyfive/footer"} /-->
```

Note that this is a normal (not synced) pattern. This means it literally includes the pattern's block markup in the parts/footer.html file, which is then included in the template files. More on theme-registered patterns later.

Customizing Templates and Template Parts

Remember how I said the template hierarchy for block themes is the same as for classic themes? I lied. Block themes have another layer to the template hierarchy - the database.

For Full Site Editing to be useful, the user needs to be able to modify the templates. That's pretty obvious. But we've already learned that if you modified the actual template files, they'd be overwritten on the next theme update. That's where the database comes in.

Block themes allow users to modify templates in the Block Editor, just as you would write a post. This functionality is available in the Site Editor (explained later). You can literally do anything - add/remove a block, change block settings, or even build the template from scratch.

Such user-created templates are saved to the database as posts, where the post_content is the markup of the entire template (it'd be exactly the contents of the .html file if you didn't change anything). The post_type is either wp_template or wp_template_part (yes, you can modify template parts as well).

The templates and template parts in the database take priority over their file-based counterparts. This is like child themes for classic themes, except it's a layer above. The order of the template hierarchy for block themes looks like this:

1. Database

2. Child theme (if active)
3. Parent theme

Creating Templates and Template Parts

Let's say you wanted to create a theme for distribution. You could write all of your templates by hand, but that's stupid. You're going to make a mistake writing the block delimiters, you'll have to constantly look up the correct syntax, and it'll be painfully slow. A better way is using the Block Editor.

You already know the power of the Block Editor - you can create the block markup from a graphical interface. But you've also already learned that templates created this way get saved in the database, not in files, and you certainly can't give someone your database if you want them to install your theme.

In reality, you'd create your templates in the Site Editor and then manually copy the generated markup to your template files or use the built-in export functionality. Alternatively, you could use the official Create Block Theme plugin. This plugin is more complex, and allows you to export not only templates but other modifications stored in the database as well. You should check it out if you're planning on creating and distributing a block theme.

Custom Templates

We've already talked about custom templates in the classic themes section. They were called 'Page Templates' back then. The idea in block themes is the same. You create a template that the user can manually choose for a post/page. This chosen template is then at the top of the template hierarchy.

The difference between custom templates in block themes and classic themes (except obviously for the file type) is how they are registered. With custom themes, you had to put a comment in the PHP file. With block themes, you have to register the template (the HTML file) in the theme.json file. We'll talk about it more when covering this mysterious file.

Just like normal templates, users can create new or overwrite custom templates in the Site Editor. This version will be stored in the database with post type wp_template and will have priority over the file.

Patterns

We've already talked about Block Patterns while discussing blocks. We didn't go into much detail back then, only what they are (pre-arranged groups of blocks), types of patterns (synced vs not synced) and how to register them. This section is focused on the more concrete ways of using patterns when building websites and themes.

Patterns in block themes are usually registered by placing them in the /patterns directory. As explained in the aforementioned section, the metadata about the pattern is provided in the

header comment. These patterns are PHP files. Note that they can only be normal (not synced) patterns. Registering synced patterns by themes is not possible (for now).

Why are patterns PHP files and not HTML files? Well, there are a couple of benefits to using PHP:

- Strings become translatable (crucial for hard-coded, theme specified strings).
- You can use PHP functions and dynamic data.
- You can specify the options in a PHP comment for WordPress to auto-register the pattern.

Unlike templates and template parts, theme-registered patterns cannot be edited directly by the user. If the user wants to modify a pattern, they have to create a new one by duplicating the one they want to edit.

Using Patterns In Templates

Remember the footer.html template part in the Twenty Twenty Five theme? It used a twentytwentyfive/footer pattern as its only content. Let's look at it again.

```
<!-- wp:pattern {"slug":"twentytwentyfive/footer"} /-->
```

By the way, the pattern block is only meant to be used in code by theme developers. You will not find it in the visual Block Editor. What this block does is it just loads the pattern registered under the specified name. Note that, because it's not a synced pattern, it'll be loaded as a dumb copy. You will not see the wp:pattern delimiter if you switch to the Code editor view. You will see the markup of the pattern.

Let's assume the footer pattern was a very simple pattern, having only a static image. This is what the /patterns/footer.php source code might look like:

```
<?php
/**
 * Title: Footer
 * Slug: twentytwentyfive/footer
 * Categories: footer
 * Block Types: core/template-part/footer
 * Description: Footer with just an image.
 */
?>

<!-- wp:image {"sizeSlug":"full","linkDestination":"none"} -->
<figure class="wp-block-image size-full">
<img src="<?php echo get_template_directory_uri(); ?>/assets/images/sample-image.png"/>
```

```
</figure>
<!-- /wp:image -->
```

As you can see, we're using the get_template_directory_uri() WordPress PHP function to get the URL of the image. This is good, as we wouldn't be able to do that without PHP. Keep in mind when this code is actually executed.

The PHP code is compiled when the pattern is registered, and that happens on the init hook. At this point, WordPress is yet to have many important properties set up, such as the global $wp_query or $post objects. This means you can't use functions that depend on that data such as the_content() or is_single() in patterns. The final HTML is compiled now, and only inserted later when using the pattern.

The practical consequence of that is that patterns cannot be used for dynamic, page-specific content. That's because by the time WordPress gets to handling the specific page, the pattern's HTML is already rendered and stored in memory.

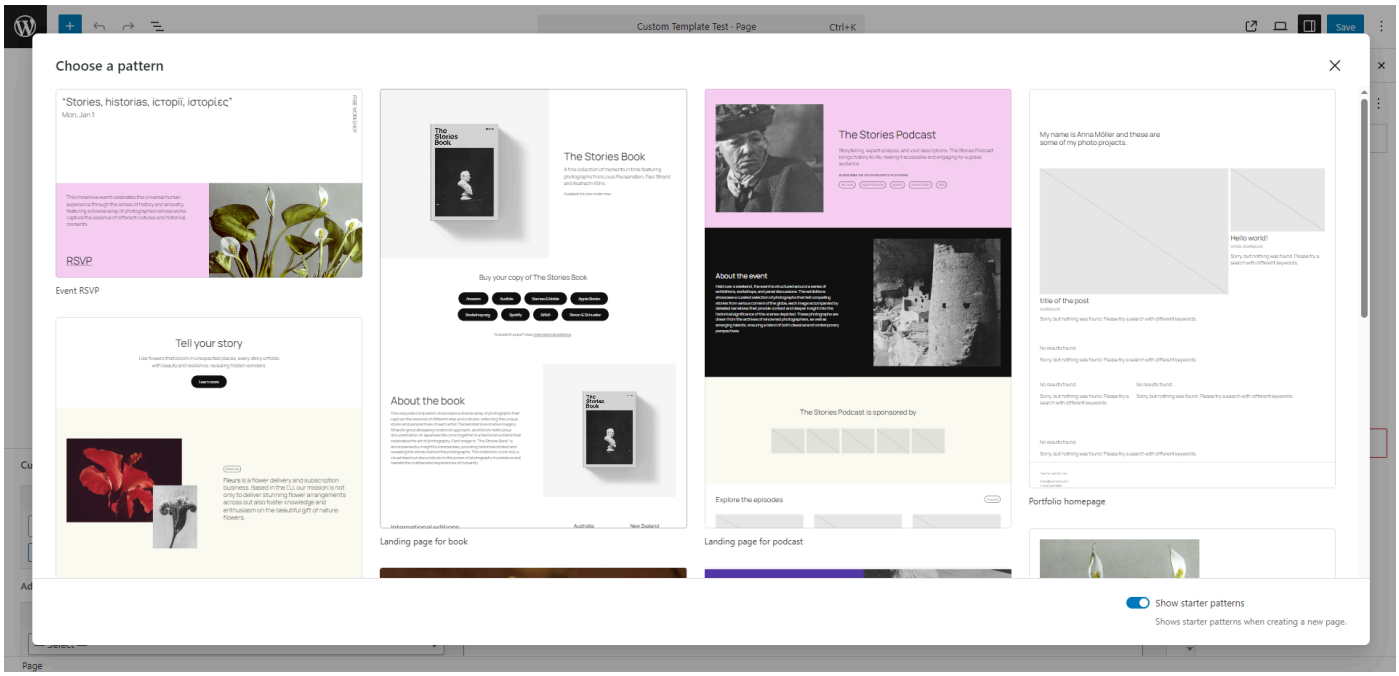
An interesting side effect of using normal patterns in template files is them acting a little like synced patterns. Just imagine you use some theme-registered pattern in 5 of your templates. If you change the underlying pattern, all of the templates are going to change as well, even though the pattern is not synced.

This happens because the pattern is re-compiled every time the template is rendered from the file. Of course this magic stops working the moment the user modifies the template in the Site Editor, as this saves the rendered HTML in the database.

You can still make it work if you put the pattern block in a template part and the user doesn't modify the template part itself (like the Twenty Twenty Five theme did with the pattern being placed in the footer template part). Unlike the pattern block, the template part block doesn't have to be rendered before it gets saved in the database. Think about it for a second, it's a nice exercise to see if you really understand how block parsing works ;).

Starter Patterns

If you create a new page (or post), or the page you open to edit has no content, WordPress will, by default, show you starter page patterns to choose from. These are full, pre-assembled, pre-styled pages that you can add to your page in one click instead of having to start from a blank canvas. Here's what that looks like:



Any pattern can be registered as a page pattern. It requires a specific set of parameters when registering the pattern (the header comments):

- **Block Types** - has to contain 'core/post-content'.
- **Post Types** - allows you to specify post types for which the pattern will be suggested.

If you do that, your pattern will be displayed in this modal every time a user creates a post of the selected post type(s). Besides starter page patterns, there are also starter template patterns. They work exactly like page patterns, but they are suggested when the user creates a template in the Site Editor (e.g., a single post, an archive, or a front page template).

To make your pattern a template pattern, you have to specify the 'Template Types' parameter when registering the pattern. This attribute sets the template types for which the pattern will be suggested, e.g., "index", "home", "single", "singular", etc.

Block Type Patterns

Block type patterns are patterns linked to a block. This linking is done by using the 'Block Types' property. We've linked our starter page patterns to the core/post-content block. But what is it? What does it mean to link a pattern to a block?

Linking, or connecting, a pattern with blocks, just means you make WordPress aware that the pattern is meant for the blocks. In practice, that means you can choose to transform a block into this pattern when inserting it.

When you insert a block that has linked patterns, you can click on its icon and choose a pattern. This plain, unstyled block, will then be replaced with the chosen pattern. How does it differ from just using the pattern in the first place? Well, the result is the same.

The power of block type patterns is that you don't have to remember all of the patterns and search for them. You create some patterns for the grid block, and then you choose the right pattern when inserting a grid - or you don't choose a pattern at all. A block which benefits a lot from that is the Query Loop block. You can have pre-defined types of loops and just choose the one you want when inserting the block.

theme.json

theme.json is a global theme configuration file. It tells WordPress what settings you want to enable, provides styles for specific elements, lets you register templates and template parts, and more. It's a step up from having to configure your theme with function calls and hooks in the functions.php file.

There are 7 top-level properties that can be configured. Here's what the "empty" file looks like:

```
{
	"$schema": "https://schemas.wp.org/trunk/theme.json",
	"version": 3,
	"settings": {},
	"styles": {},
	"customTemplates": {},
	"templateParts": {},
	"patterns": []
}
```

- **$schema** - defines supported JSON schema. Can be used by code editors to provide hints and error reporting.
- **version** - the theme.json schema version. The latest version (as of August 2025) is 3. The theme.json Living Reference is the most up-to-date document on theme.json.
- **settings** - defines block controls, color palettes, layouts, font sizes, and more.
- **styles** - applies styles to the website and blocks.
- **customTemplates** - metadata for custom templates.
- **templateParts** - metadata for template parts.
- **patterns** - allows you to bundle patterns from the Patterns Directory with your theme.

Bear in mind, theme.json is a huge topic. There are dozens if not hundreds of different properties you can configure. Listing and explaining all of them in detail in this guide would be neither realistic nor helpful. The following list is limited to only the most important properties and values. Read the official theme.json documentation and the aforementioned Living Reference if you need a more in-depth view.

settings

The settings property configures, as the name implies, settings. Let's look at the top level structure first and break some of it down later.

```
{
	"version": 3,
	"settings": {
		"appearanceTools": false,
		"background": {},
		"border": {},
		"color": {},
		"custom": {},
		"dimensions": {},
		"layout": {},
		"lightbox": {},
		"position": {},
		"shadow": {},
		"spacing": {},
		"typography": {},
		"useRootPaddingAwareAlignments": false,
		"blocks": {}
	}
}
```

You might be a little confused here. Most of these properties sound like they should be placed in the styles property instead. An important thing to understand is that the settings properties are usually responsible for either enabling (and disabling) functionality in the editor or registering presets (e.g., color presets). These settings provide the tools, and the tools are then utilized by users and styles.

Most properties in settings are boolean properties. They are either true or false and they control the behavior of the website in some way. Let's look at some (not all) of these properties so that you get the idea:

- background.backgroundImage - allow users to set a background image.
- border.color - allow users to set custom border colors.
- border.radius - allow users to set custom border radius.
- color.background - allow users to set background colors.
- color.text - allow users to set text colors in a block.
- color.custom - allow users to select custom colors (with a color picker instead of a predefined set of colors).
- dimensions.minheight - allow users to set custom minimum height.
- position.sticky - allow users to set sticky position.
- spacing.margin - allow users to set a custom margin.

- spacing.padding - allow users to set a custom padding.
- spacing.customSpacingSize - allow users to set custom space sizes.
- typography.customFontSize - allow users to set custom font sizes.
- typography.fontStyle - allow users to set custom font styles.
- typography.letterSpacing - allow users to set custom letter spacing.

These are only a few of the properties defined in theme.json v3. It's important to understand what really happens when you set one of them. Let's take position.sticky as an example. Just because you set it to true doesn't mean every block will now have the 'sticky' checkbox. It only signals to WordPress that it can show this checkbox.

This difference is crucial so it's important you pay attention. The block author defines what attributes and options the block supports. You should know that already if you read the blocks section. The theme author, using theme.json, can enable or disable support for certain options.

If the block supports position.sticky, but the theme disabled it, it won't be shown. If the block doesn't support position.sticky, and the theme enabled it, it still won't be shown. It will only be shown if the block supports position.sticky and the theme enables it.

It's important to remember that most of these properties have some default values. This will be 'true' for some and 'false' for others. Consult the documentation for more info. Let's now look at a few important, non-boolean properties.

color.palette, color.duotone, and color.gradients let you register custom color presets. Palette is for normal colors, duotone are dual colors, usually used for image filters (shadow and highlight), and gradients are gradients. These properties expect an array with a 'name', 'slug', and some values. Here's an example:

```
{
      "version": 3,
      "settings": {
            "color": {
                  "palette": [
                        {
                              "color": "#ffffff",
                              "name": "Base",
                              "slug": "base"
                        },
                        {
                              "color": "#000000",
                              "name": "Contrast",
                              "slug": "contrast"
                        },
```

```
                ],
                "gradients": [
                        {
                                "gradient": "linear-gradient(to right, #10b981,
#64a30d)",
                                "name": "Emerald",
                                "slug": "emerald"
                        }
                ],
                "duotone": [
                        {
                                "colors": [
                                        "#450a0a",
                                        "#fef2f2"
                                ],
                                "name": "Red",
                                "slug": "red"
                        }
                ]
        }
    }
}
```

This triplet of {value, name, slug} is common in settings. You will see it in many other properties such as border.radiusSizes, dimensions.aspectRations, shadow.presets, and more. All of these allow you to register presets.

layout.contentSize is an important property. It sets the max-width of content. It is typically used for controlling the width of the post content and other similar areas on the page. This should be set to a value which makes the content readable (a line of text should usually be between 45 and 75 characters wide).

spacing.units is another important property. It expects an array of strings and allows you to specify supported units for spacing-related attributes. The default value is ["px","em","rem","vh","vw","%"].

spacing.spacingScale lets you define a custom spacing scale. What's a spacing scale? It's just a way of defining a spacing system (spacing presets). I imagine that didn't clarify much… Let's start from the beginning.

Many web designers use a standard scaling system for spaces. These are just pre-defined values for spaces expressed as steps, such as: 4px, 8px, 16px, 24px, 40px, 64px, etc. The

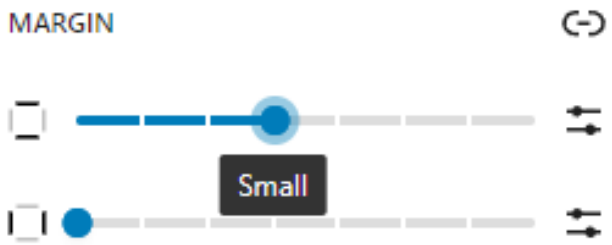
designer then doesn't apply a random number when choosing a margin for an element, but instead picks from this pre-defined set of spaces. That's a spacing system.

spacing.spacingScale has 5 properties:

- operator - either + (addition) or * (multiplication). Defines how the steps get calculated (default *).
- increment - a number used with the operator to calculate the next step (default 1.5).
- steps - the total number of steps in the scale (default 7).
- mediumStep - the medium (center) value of the scale (default 1.5).
- unit - a CSS unit (default rem).

The spacing system is then automatically generated using these values. What really gets generated is CSS variables following this naming convention: --wp-preset--spacing--{step}. Steps are incremented by 10. The mediumStep defines what values the rest of the steps are going to have. Let's look at what's generated with the default values.

--wp-preset--spacing--50 is the middle step. --wp-preset--spacing--60 (the step after) is calculated by multiplying 1.5 rem (mediumStep value) by 1.5 (increment value). That gives us 2.25 rem. --wp-preset--spacing--70 is then calculated by doing 2.25 rem * 1.5 = 3.38 rem. Similarly, --wp-preset--spacing--40 is calculated by doing 1.5 rem / 1.5 = 1 rem. We're going the other direction from the middle so we have to divide instead of multiplying.



And here's what these steps then look like in the Block Editor. They are sliders allowing you to choose one of the steps. Note that the middle step will always be 50, and the lowest step is 10. This means that if you have more than 10 steps, they will be cut off on the lower end of the scale. Couple that with the limiting nature of mathematically generated systems, and you might sometimes need something more precise.

spacing.spacingSizes allows you to specify all of the steps by hand. Here's an example:

```
{
	"version": 3,
	"settings": {
```

```
            "spacing": {
                  "spacingSizes": [
                        {
                              "name": "Step 1",
                              "size": "4px",
                              "slug": "10"
                        },
                        {
                              "name": "Step 2",
                              "size": "8px",
                              "slug": "20"
                        },
                        {
                              "name": "Step 3",
                              "size": "16px",
                              "slug": "30"
                        },
                        {
                              "name": "Step 4",
                              "size": "24px",
                              "slug": "40"
                        }
                  ]
            }
      }
}
```

This will, once again, generate --wp-preset--spacing--{step} variables. The step in this case is the slug property. The name is the label used for the step (you could see "Small" in the screenshot above). You're also not limited to incrementing your slug by 10 or even making it a number (although both of these are good practice).

The **custom** property is a unique property which allows you to create custom CSS variables. These variables follow this naming convention: --wp--custom--{key}, where key is transformed to kebab-case. It makes the most sense to start with an example:

```
{
      "version": 3,
      "settings": {
            "custom": {
                  "lineHeight": "1.4"
            }
```

```
	}
}
```

This will create the following CSS:

```
body {
	--wp--custom--line-height: 1.4;
}
```

Custom variables can also be nested indefinitely. This means you can do:

```
{
	"version": 3,
	"settings": {
		"custom": {
			"lineHeight": {
				"xs": "1",
				"sm": "1.25",
				"md": "1.5",
				"lg": "1.75"
			}
		}
	}
}
```

And it will create this CSS:

```
body {
	--wp--custom--line-height--xs: 1;
	--wp--custom--line-height--sm: 1.25;
	--wp--custom--line-height--md: 1.5;
	--wp--custom--line-height--lg: 1.75;
}
```

The **blocks** property is another unique property. All of the settings we've been talking about were global settings. They affected all of the blocks. The blocks property lets you modify settings for specific blocks. For example, you could create specific color presets for the button block by setting the "settings.blocks.core/button.color.palette" property. These settings overwrite the global settings (the colors will overwrite the global colors specified with settings.color.palette).

styles

The styles property is the central place for styling block themes. That's in stark contrast with classic themes, which were styled using style.css or other CSS files. If you can, you should style your theme this way (in theme.json). This allows the user to modify those styles in the Site Editor (explained later).

The pre-defined styles properties are direct counterparts to the most used CSS properties. If the property you need doesn't exist, you can use the css property to write the CSS by hand. The styles property lets you target 3 types of entities: root, elements, and blocks.

Root is the <body> element. All styles applied directly to the styles property target the root. Elements are HTML elements. The currently supported elements are: button (<button> and button-like links), caption (<figcaption>), cite (<cite>), heading (any <h{x}>), h1-h6 (individually), and link (<a>). Blocks let you style individual blocks.

Here's what an "empty" styles property might look like:

```
{
	"version": 3,
	"styles": {
		"elements": {},
		"blocks": {}
	}
}
```

Let's assume we want to set the background color of the entire website to #fefefe. We would do that on the <body> element. We also want all buttons to have a red background and a white text. Lastly, we want our image blocks to have slightly rounded corners. Here's what our theme.json file might look like:

```
{
	"version": 3,
	"styles": {
		"color": {
			"background": "#fefefe"
		},
		"elements": {
			"button": {
				"color": {
					"text": "#ffffff",
					"background": "red"
				}
```

```
                }
            },
            "blocks": {
                "core/image": {
                    "border": {
                        "radius": "6px"
                    }
                }
            }
        }
}
```

This will automatically generate all the necessary CSS. You can also style pseudo-classes, such as :hover. To do that, just use the pseudo-class after the element, such as "styles.elements.button.:hover.color" to change the color of the button on hover.

You can style elements nested in blocks. That is - if you have a block which has a <button> element in its markup, you can target only the button element by doing: "styles.blocks.block/name.elements.button". You basically have to put the elements property inside the block property.

Remember Block Styles from the section about blocks? You had to register them either in PHP or in JS, and they added a Styles select in the block editor. Selecting a style there resulted in an 'is-style-{name}' CSS class being added to the block. You can style those in theme.json as well. All you have to do is add a variations property to the block. So to style the 'outline' Block Style of the button block, you'd have to target styles.blocks.core/button.variations.outline.

In the example above, we used hard-coded values. This is not how you'd usually write your styles. You'd usually use presets. You already know presets. You've registered them in the settings property. When creating the spacing system - you registered presets. When defining the color palette - you again registered presets.

All presets from the settings property create a corresponding CSS property (variable). This is --wp-preset--spacing--50 for the spacing step, and --wp-preset--color--base for a color with slug "base". The general rule is --wp-preset--$feature–$slug.

To use presets in the styles properties, you should follow a specific syntax. To use the base color, you would write "var:preset|color|base". This will use the --wp-preset--color--base variable. You could technically use the CSS native var(), but the WordPress way is preferred. This means we could do:

```
{
```

```
	"version": 3,
	"styles": {
		"background-color": "var:preset|color|base",
		"fontFamily": "var:preset|font-family|default"
	}
}
```

What about custom CSS variables (defined in the custom property)? These weren't presets. The final CSS variable looked like this: --wp--custom--line-height--lg. You can reference those by doing "var:custom|line-height|lg". Note that you can't reference any arbitrary CSS variable using this notation. Only "var:preset" and "var:custom" are defined.

customTemplates

I promised we'll talk about registering custom templates when we cover theme.json. Well, here we are. Just to remind you, these are templates for pages and other post types that the user can manually select for the post.

It's not anything hard, and it's certainly not going to be surprising to you if you've been following this guide meticulously. Here's the code:

```
{
	"version": 3,
	"customTemplates": [
		{
			"name": "2-columns-layout",
			"title": "2 Columns Layout",
			"postTypes": [
				"page",
				"post",
				"book"
			]
		}
	]
}
```

The name property is the name of the file, i.e., /templates/2-columns-layout.html.

templateParts

You may be thinking "But I thought you didn't have to register a template part???". You are correct - you don't, but you can. Here's what it looks like:

```
{
	"version": 3,
	"templateParts": [
		{
			"area": "footer",
			"name": "footer-2-columns",
			"title": "Footer 2 Columns"
		},
	]
}
```

Okay, so what the hell is this? Well, the name is the file name. It means the part is located at /parts/footer-2-columns.html. The title is displayed in the Site Editor. This is where the first benefit of registering parts becomes apparent. If you were to not register your part but only placed them in the /parts folder, the name of this part shown to the users would be "footer-2-columns".

But what is 'area'? In the Site Editor, template parts are categorized by areas. That's literally it, a category for parts. There are 3 areas defined by default: footer, header, and uncategorized. You can register custom areas, but that's advanced and pretty niche so I won't cover it. The area doesn't affect the part at all, only how it's displayed in the admin panel (in the Site Editor).

Style Variations

Perhaps a more fitting name for this section would be "theme.json variations". WordPress lets you provide multiple versions of the theme.json file. These versions can provide vastly different styling and behavior of your theme. Let's see how that works.

The main theme.json file should be placed in the root of your theme folder. This file will be the default configuration file of your theme the first time the user activates it. Style variations are other .json files. These files should be placed in the /styles folder.

They shouldn't be named theme.json, but they can contain every 'settings' and 'styles' property theme.json can. So why do they exist? They are alternative theme.json files your users can select in the Site Editor.

When the user selects one of your style variations, the options defined in that JSON file are stored in the database. They are then used in place of the default theme.json configuration. This allows you to provide multiple different styles bundled with your theme. Let's say you created a theme for restaurants. You could provide a theme.json file optimized for cafes, one for pubs, and one for high-end restaurants.

Note that the original theme.json file is still used. If a property is defined in theme.json but is not defined in a style variation, it'll be used even if the style variation is selected. Style variations

can overwrite properties from theme.json and add new properties, but they don't cause theme.json to be ignored. If you want to disable some style or setting from theme.json in your style variation, you have to do it explicitly.

The last difference between style variations and theme.json is the top-level title property. You need it to define the title of the variation to be displayed in the Site Editor. Let's create an "empty" variation:

```
{
	"version": 3,
	"title": "High-End Restaurant",
	"settings": {},
	"styles": {}
}
```

You could then place this in a /styles/high-end.json file.

Site Editor

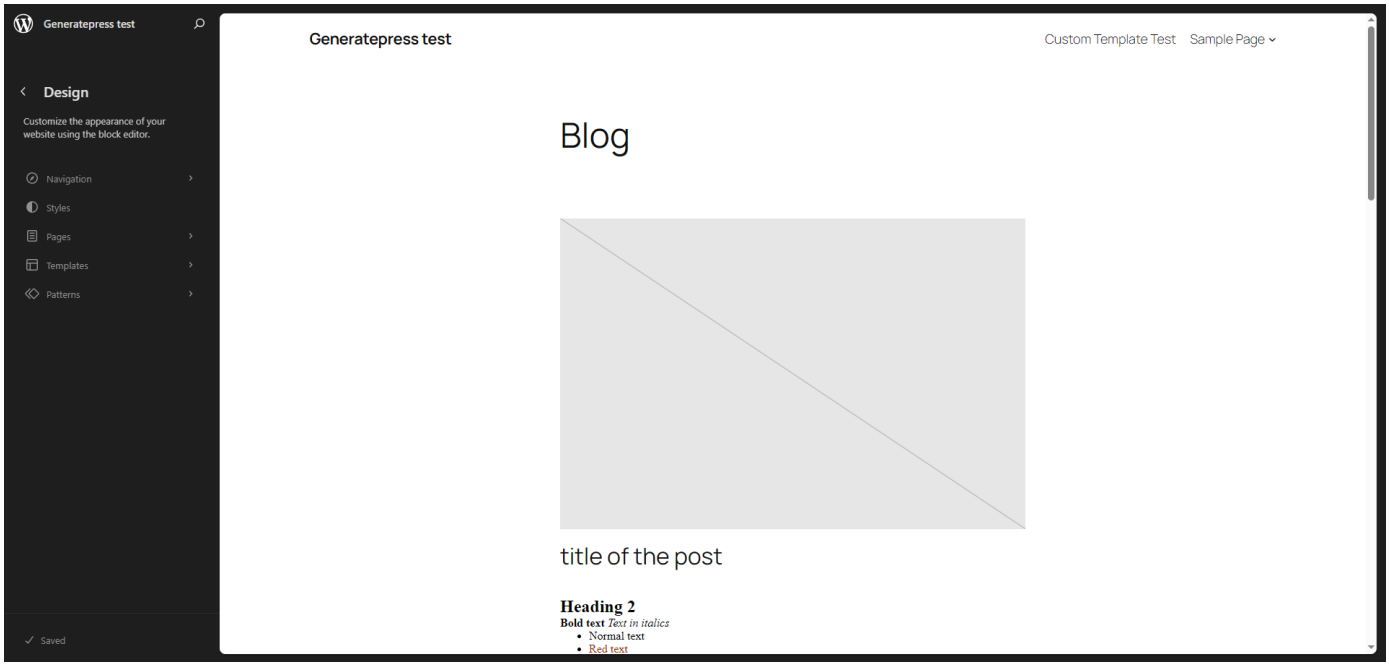
I had a hard time deciding where to put this section. It's a chicken-and-egg problem. You need the context of the Site Editor to understand other topics related to block themes, but you won't understand the Site Editor until you understand those other topics.

If you didn't fully grasp the sections that came before, there are 2 possible reasons why:

1. I suck at writing.
2. You lacked understanding of the Site Editor.

If it's the former, make sure to let me know. If it's the latter, you might benefit from quickly going over those sections again after you finish this one.

The Site Editor is the UI for Full Site Editing. That's it. It's the panel where you create/modify templates, manage navigation menus, change styles, and more. It's only available in Block Themes, and you access it by going to Appearance -> Editor in the side menu of the admin panel. Here's what it looks like:



You can see 5 top level links in the menu on the side:

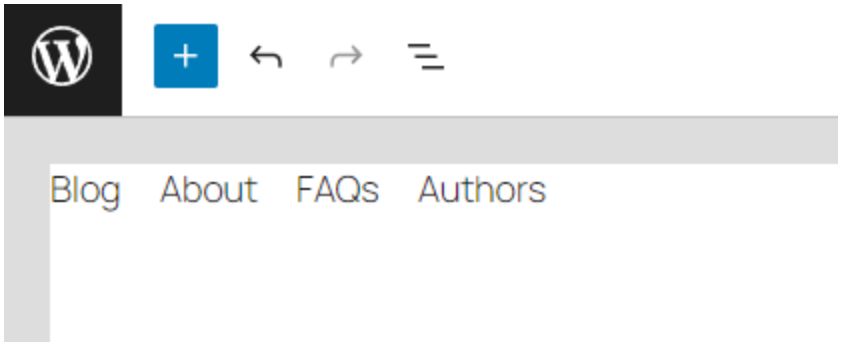
- Navigation
- Styles
- Pages
- Templates
- Patterns

Navigation

Menus in block themes work completely differently than they do in classic themes. Let's start with how they're used in templates.

To put a navigation menu in a template, you have to use the core/navigation block. When you place this block anywhere in the editor, you'll have the option to make it display one of your menus or you can create a new menu. The "navigation" tab in the Site Editor lets you do that as well. You can see all of your menus, edit them, delete them, and create new ones.

A menu is just a post. When you create a new menu, either when editing a template or from the navigation tab, you'll do so in the Block Editor. Here's what a menu looks like in the Site Editor:



The contents of the menu can only be specific blocks, such as the navigation-link block or the navigation-submenu block. You don't have to add those blocks manually from the inserter, you create the menu by typing the names and links and WordPress inserts the blocks automatically.

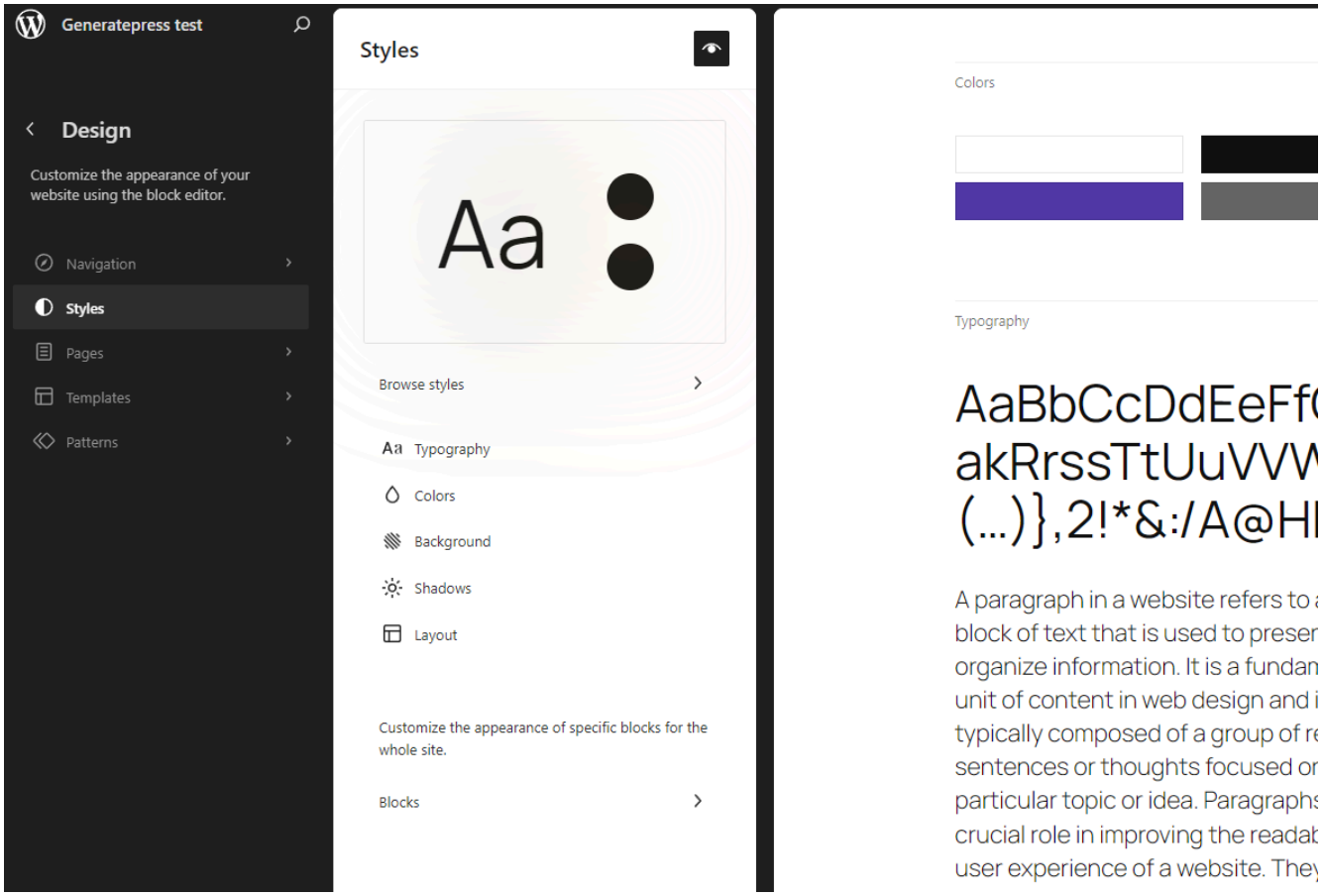
All menus are stored in the database. They are stored as posts with type wp_navigation. That's what you're editing on the screenshot above - the post_content column of this post. And here's what that post_content might look like:

```
<!-- wp:navigation-submenu {"label":"Submenu
test","type":"page","id":102,"url":"http://localhost:8001/custom-template-te
st/","kind":"post-type"} -->
<!-- wp:navigation-link {"label":"test
hierarchy","type":"page","id":75,"url":"http://localhost:8001/sample-page/te
st-hierarchy/","kind":"post-type"} /-->
<!-- /wp:navigation-submenu -->
<!-- wp:navigation-link
{"label":"customurl.com","type":"link","url":"https://customurl.com","kind":
"custom"} /-->
```

You can see those are just blocks, as you would expect. You can also see different types of blocks, and different types of links (internal page and custom link). The core/navigation blocks reference the selected menu the same way synced patterns do, with a {"ref":ID} attribute. This means that if you modify a menu, this change will automatically be reflected everywhere the menu is used.

Styles

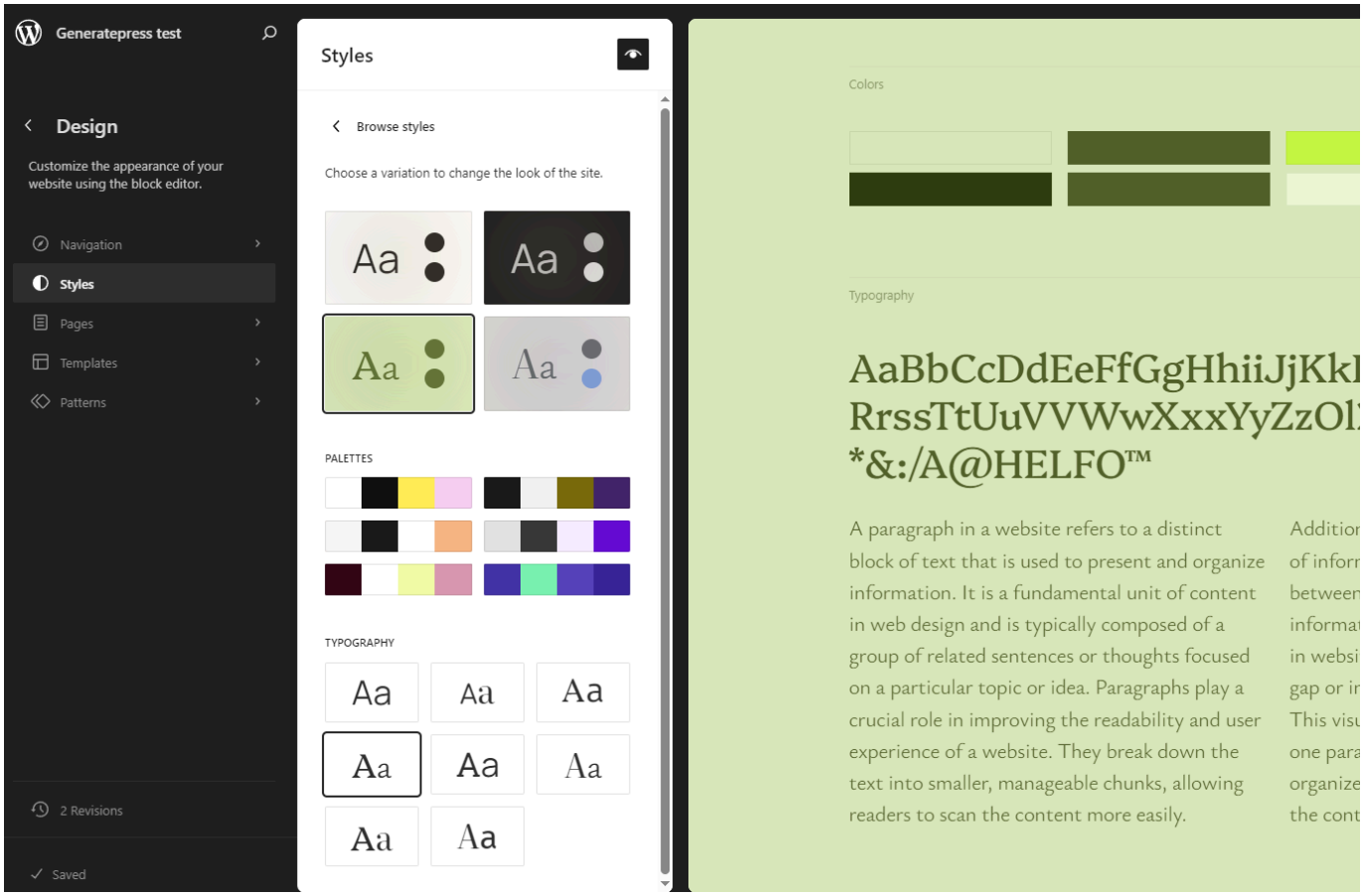
The styles tab is arguably the most important tab in the Site Editor. It's where you modify your website's global styles, typography, colors, block styles, etc. These modifications are stored in the database and take priority over the theme.json file. Keep in mind that what you see and can do here depends on the theme.json settings. Take a look at it:



There are 7 major, top level sections:

- Browse styles
- Colors
- Background
- Shadows
- Layout
- Blocks

Browse styles is where you can choose style variations. These are alternative theme.json-like files placed in the /styles folder. Here's what this panel looks like (I had to delete some variations in devtools to make them fit on the screen):



You can see the style variations at the very top of this panel. If you hover over a variation, you'll see its title. You can preview what your site will look like on the right after selecting a variation.

If you're perceptive, you might be wondering - "What are those palettes and typography sections?". This is exactly what I was wondering, and it turned out to be a very interesting rabbit hole. I must start with stating that what I'm about to describe is not documented at all (at the time of writing). It's cutting edge WordPress functionality, and I had to discover it by experimenting with the Twenty Twenty-Five theme.

If you place a JSON file in the /styles directory and that file contains only color-related settings and styles, the file will become a palette variation. Same thing for typography - a file containing only typography-related properties becomes a typography variation. The moment you add any other property, the file becomes a full style variation.

WordPress seems to compare color and typography styles present in style variations with the available palette and typography variations. If a style variation has exactly the same color properties as a palette variation, when you select the style variation, the palette variation will also be automatically selected (a dark border will appear). The same goes for typography variations.

It's hard to explain in words. When you select style variation A, the palette variation A and typography variation A are automatically marked as selected. That's of course assuming the 'color' and 'typography' properties in style variation A match 1:1 those in the palette and typography JSON files. Btw, that system does *not* follow the DRY principle.

Palette and typography variations don't have to match any style variations. They can be standalone styles for color and typography. In that case, when you select a palette/typography variation that does not match any style variation 1:1, no style variation will be marked as selected (have a dark border).

Do you think that's confusing? Well, fasten your seatbelt. Let me ask you a question. Where do you think color and typography-unrelated configuration (e.g., spacing) is derived from when you select a palette variation that doesn't match any style variation (i.e., no style variation is marked selected)? Does it fall back to theme.json?

The answer to that question is that it takes the last selected style variation. That means, if you:

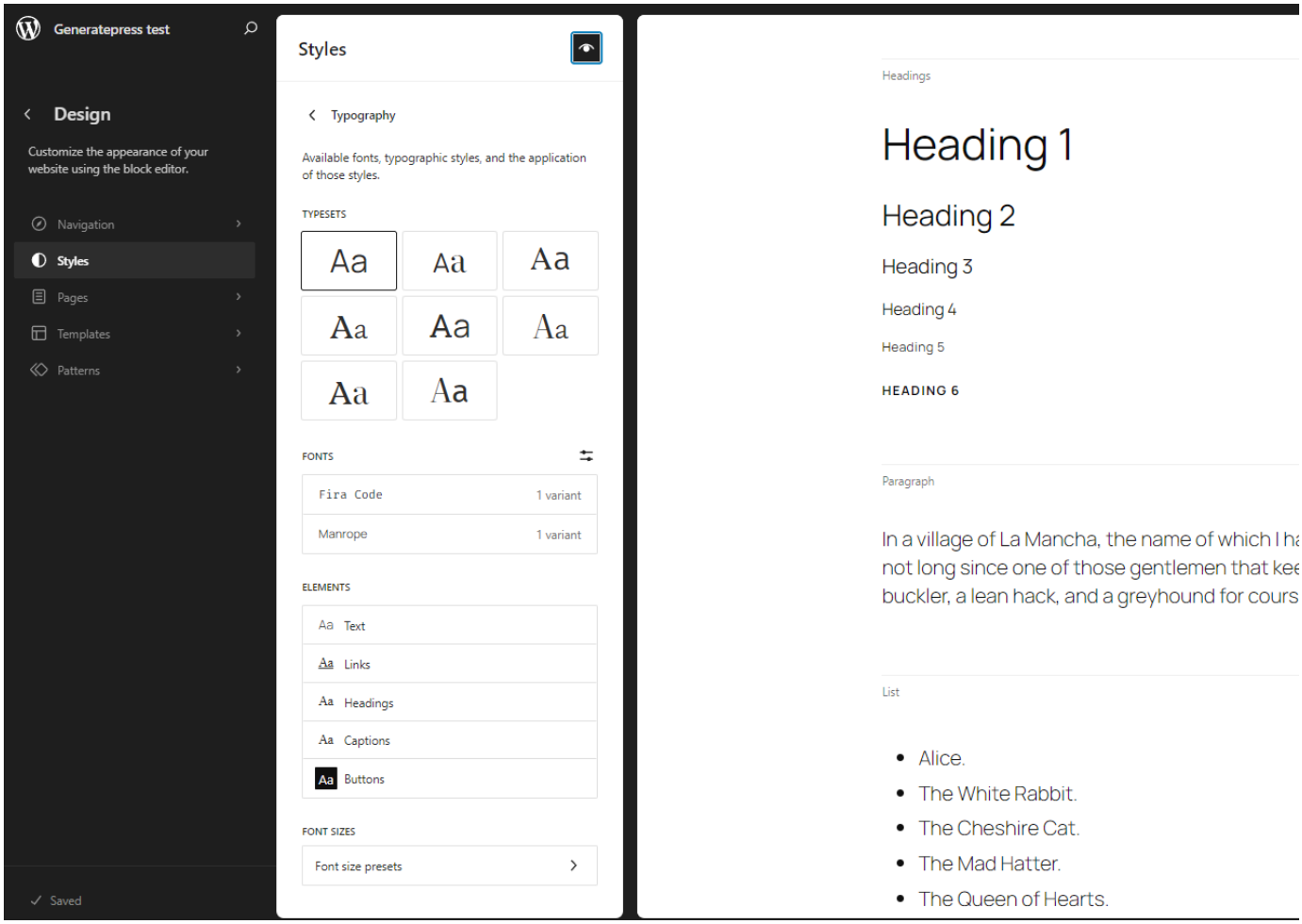
1. Select style variation A
2. Select palette variation B (causing style variation A to be deselected)
3. Save

The spacing configuration from style variation A will be used, even if it isn't visually marked as selected in the Site Editor.

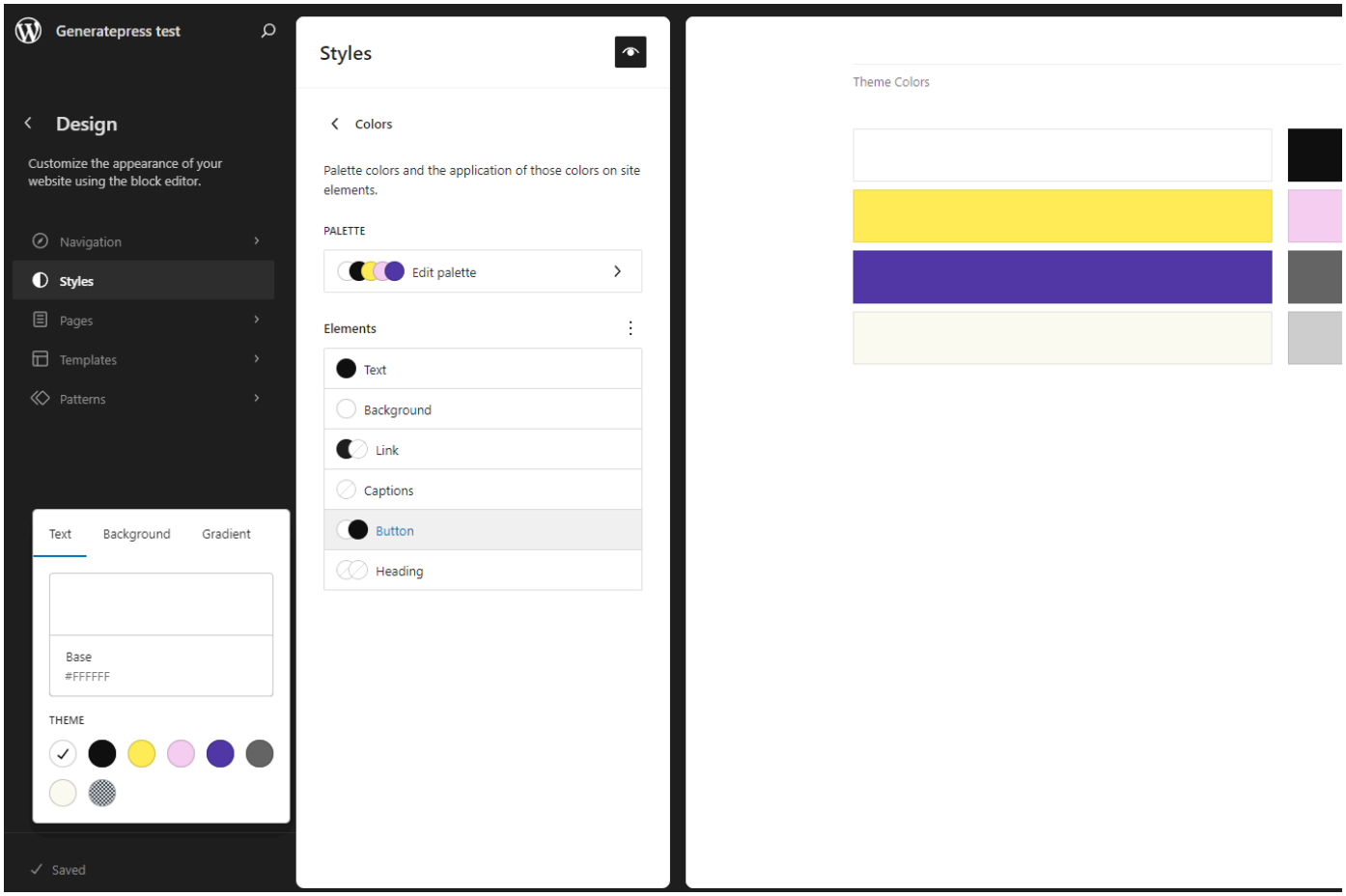
To add fuel to the fire, while playing with this system, I noticed that some palette variations disappeared and reappeared as I clicked on different style variations. I hope they write the documentation soon…

The **typography** tab allows you to modify typography-related settings. You can see the same typography variations here as we did in 'browse styles'. You can do many things here, such as:

- Choose a typography variation.
- Upload your own fonts.
- Modify typography settings for text, links, headings, captions, and buttons elements. Those are properties like font family, font size, line height, letter spacing, etc.
- Modify existing and add custom font size presets.



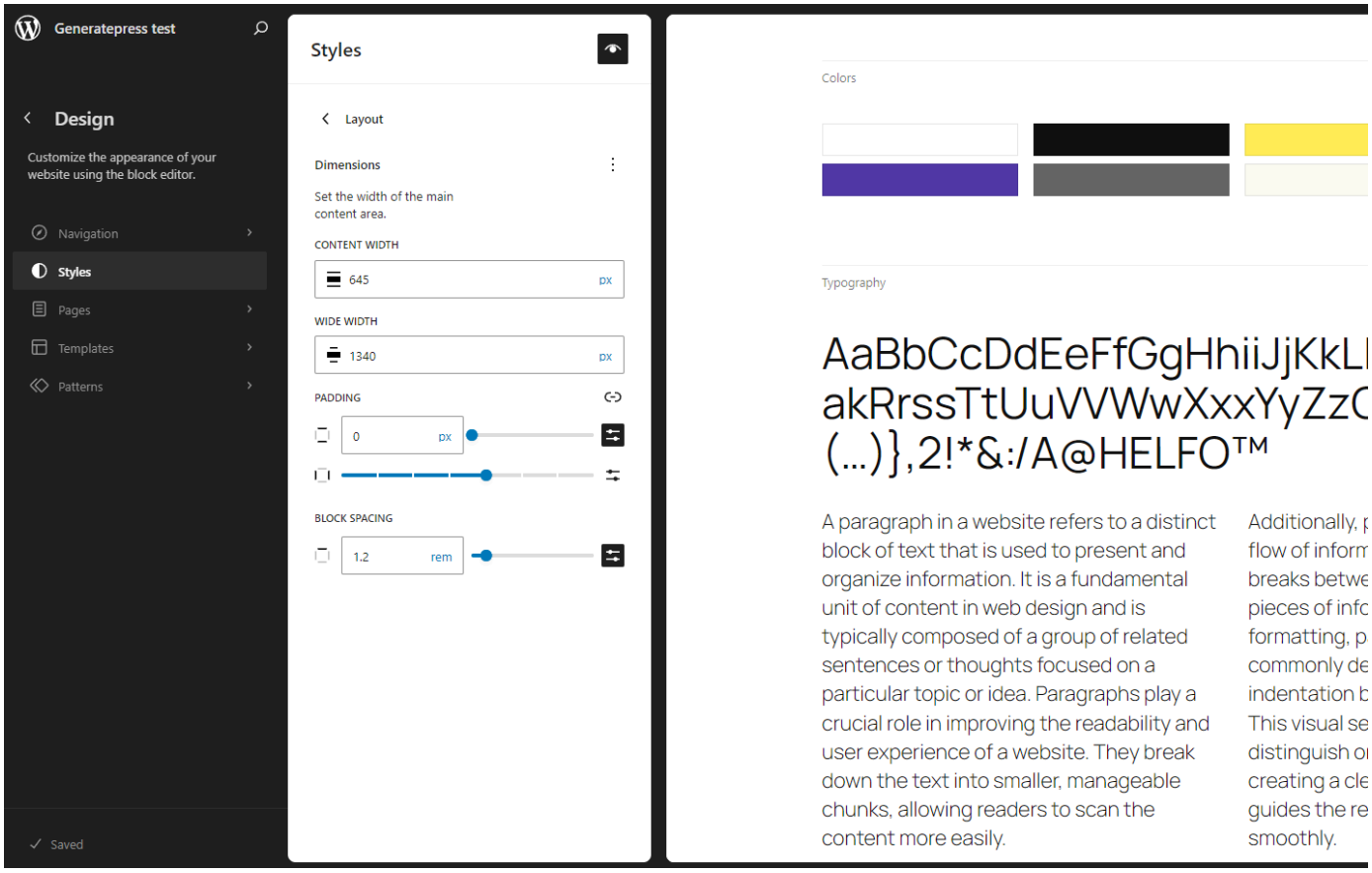
The **colors** panel lets you modify the active color palette, select a different palette variation, and change colors for text, background, link, captions, button, and heading elements.



The **background** tab allows you to set a background image (not worth a screenshot).

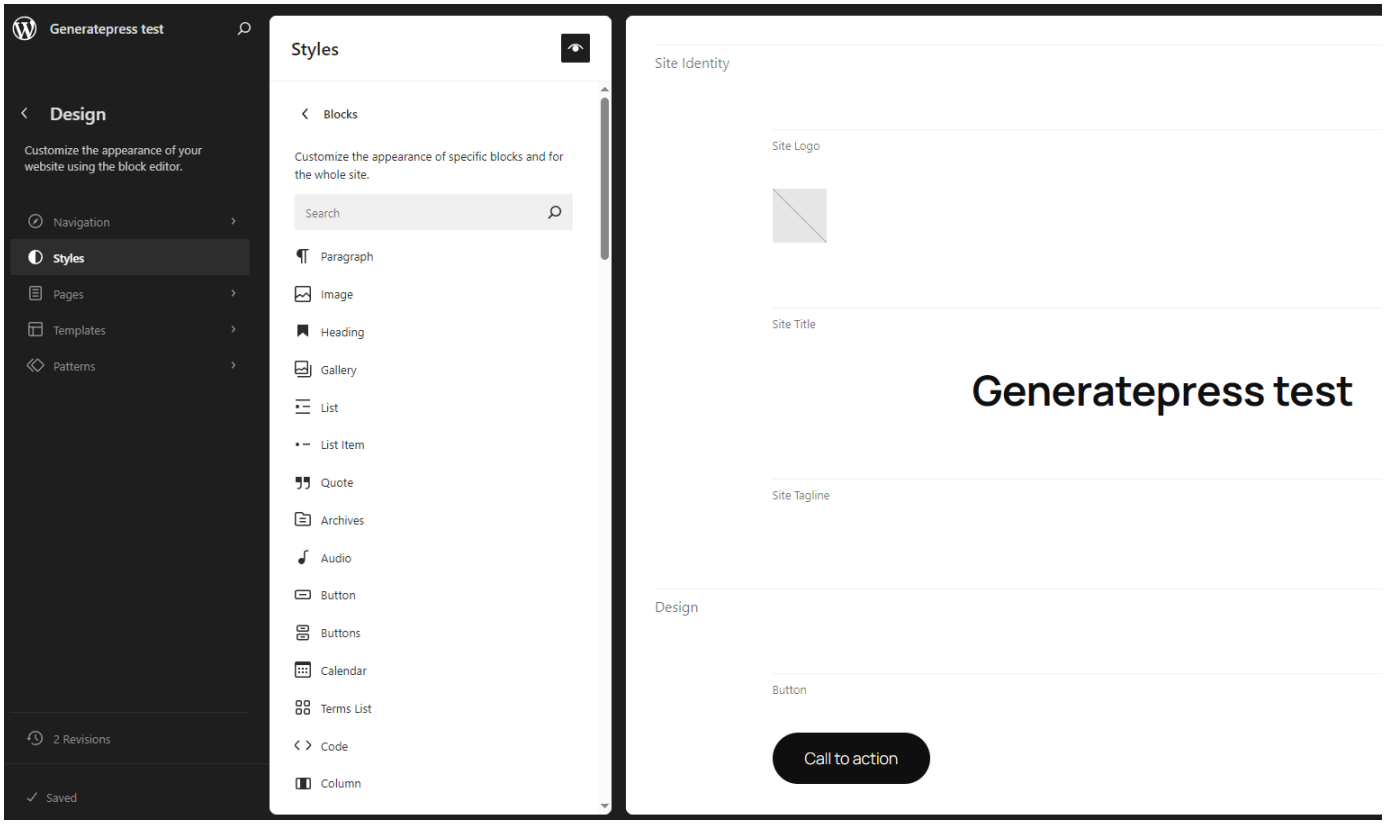
The **shadows** tab allows you to manage existing and create custom shadow presets (not worth a screenshot either).

The **layout** tab lets you control different properties related to the website's layout. You can see the layout.contentWidth property we covered when talking about theme.json.



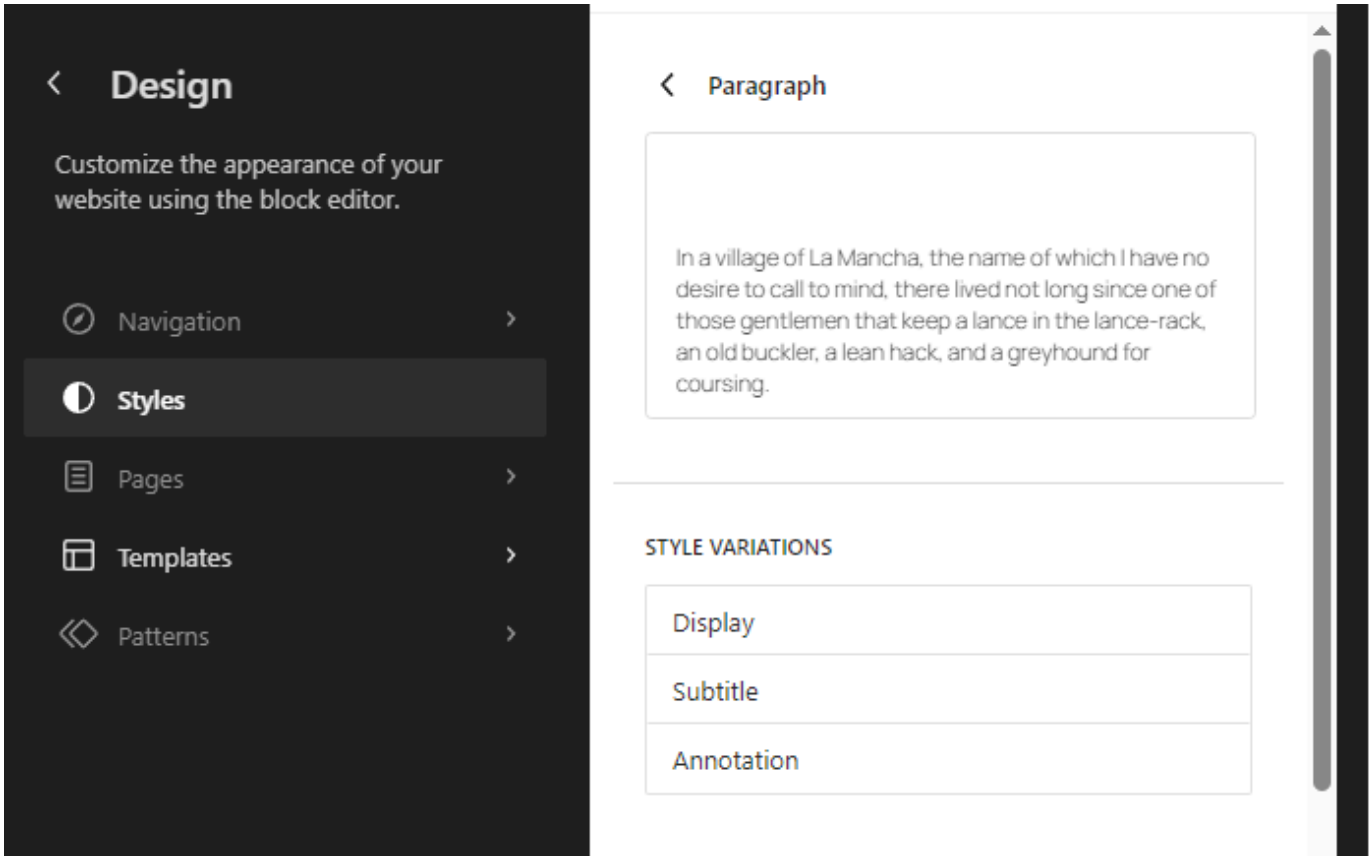
The **blocks** panel is a very useful part of the Site Editor. It allows you to modify the default style of some blocks. I don't know for certain which blocks do and don't show up here, but I've observed that blocks added with some plugins do not. Perhaps it's an opt-in feature configured in the block.json file (either explicitly or implicitly). More testing would be required to get a definitive answer.

The styling options available for each block seem to match the styling options available directly in the Block Editor. Remember that you're changing the default styling here. Any actively used blocks dependent on that styling will change, and any blocks with in-editor modifications to the updated properties will stay the same.



There's one more thing we have to discuss here. If you clicked on any individual block, you could see a "style variations" section. The terminology in WordPress is very confusing, so I'll remind you what those are.

Block style variations are the different styles you can create for blocks that the user can select with one click in the editor (e.g., rounded for a button). They are registered with PHP register_block_style() or with JS and result in "is-style-{name}" CSS class being added to the element.



If you remember the section on block styles, we said their authors were responsible for manually styling them and enqueueing their CSS files. But if a block style is styled directly with CSS, then it doesn't go through the WordPress styling system. And if it doesn't go through the WordPress styling system, how come we modify it in the Site Editor?

We're once again entering undocumented territory. The following description is based on my observation and testing of the Twenty Twenty-Five theme.

Remember when I said block styles can only be registered with PHP and JS? That was a lie. Turns out they can also be created with JSON files. A JSON file placed in the /styles directory with a 'blockTypes' property becomes a block style.

The blockTypes property defines the blocks for which the style is to be registered. You should also specify a slug and a title. The rest of the file follows the theme.json notation you already know, and you should use it to specify the style. Let's look at an example straight from the Twenty Twenty-Five theme:

```
{
	"$schema": "https://schemas.wp.org/wp/6.7/theme.json",
	"version": 3,
```

```
      "title": "Display",
      "slug": "text-display",
      "blockTypes": ["core/heading", "core/paragraph"],
      "styles": {
            "typography": {
                  "fontSize": "clamp(2.2rem, 2.2rem + ((1vw - 0.2rem) * 1.333),
3.5rem)",
                  "lineHeight": "1.2"
            }
      }
}
```

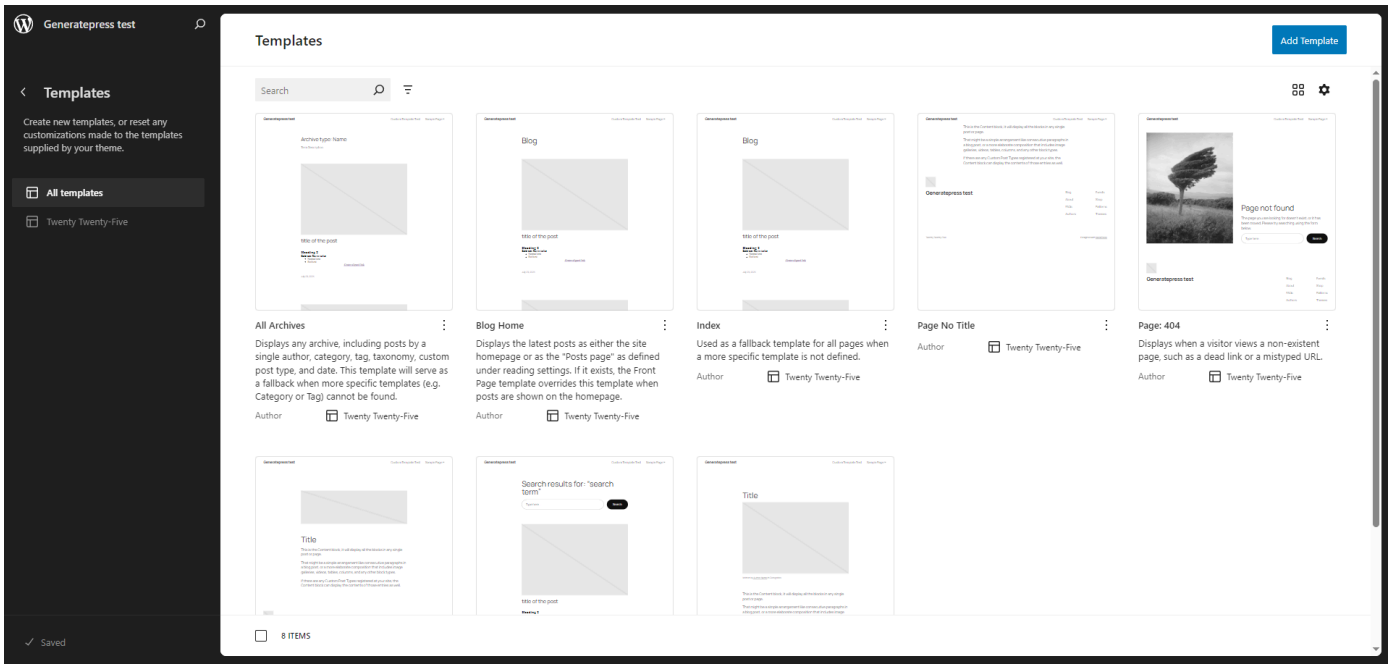
A block style created this way *does* go through the WordPress styling system, which makes it possible to unite it with the rest of the UI. Block styles registered with PHP or JS will not show up in this part of the Site Editor.

Pages

The pages panel allows you to edit pages from the Site Editor. That's it. You can view a list of all your pages, and if you click to edit one, you'll see the exact same Block Editor interface you would've if you opened it from the WordPress pages menu. I guess it was added to bring as much of the experience directly into the Site Editor as possible.

Templates

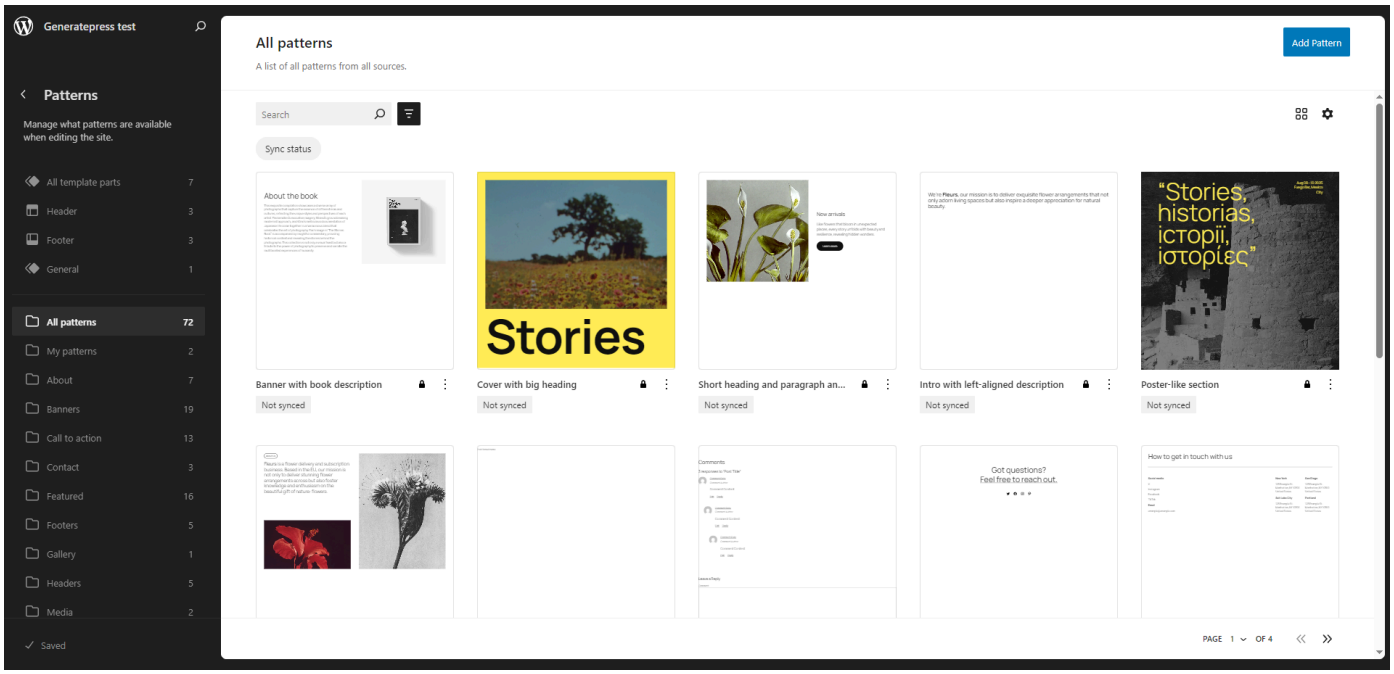
The templates panel makes it possible to modify theme templates (single.html, 404.html, archive.html, etc.). If you click on a template to modify, it'll be opened in the Block Editor. If you modify the template and save it, it'll get saved to the database (as already explained before). You can also create new templates, including custom templates.



Patterns

The patterns tab allows you to view and create patterns. It's also where you can find, modify, and create template parts.

You can see that template parts are categorized in 3 different submenus: "Header", "Footer", and "General". These are the areas we've already talked about. Similarly, patterns are categorized into many different categories, which you can choose or create when adding a pattern.



Child Themes With Block Themes

Child themes work the same, whether the parent is a block or a classic theme. The biggest difference is that their utility has greatly decreased. They were the main and only way of non-standard customizations of classic themes, but block themes solved this with FSE and changes stored in the database.

There are still a couple of valid reasons to use a child theme with a block theme:

- Adding custom CSS.
- Adding custom PHP with functions.php
- Providing a customized theme.json.

Custom CSS and PHP have not changed. You still need a child theme to create a custom CSS file, and you still need functions.php to enqueue that file. If you placed that in the parent theme, these modifications would be overridden on the next update. Nothing new here.

A customized theme.json is a little more interesting. It doesn't completely replace the parent's file. The child and parent theme.json get merged, with the properties defined in the child's version taking priority. That means you may only set the properties you're interested in modifying in your child's theme.json, and all of the rest will be inherited.

You can obviously still override templates and template parts in your child theme. This may be useful if you're handing the website to a client and don't want them to be able to clear customizations from the database. By overriding the template's file, you're providing a new default. Just remember that the database will always take priority.

The Downsides Of Block Themes

Block themes may seem pretty great, but they are not all rainbows and sunshine. They are brilliant if you're a blogger and want to have a simple, static site, but they break down on more complex projects.

The most obvious problem is the loss of direct programmatic control. Templates are not PHP files, they are HTML markup. With classic themes, if you want to display some text dynamically based on whether the user is logged in, you just use if (is_user_logged_in()). You can't use that in HTML.

Adding any dynamic data is complex. Achieving what used to take a few lines of PHP, now usually requires developing a custom dynamic block. That's a plugin you have to create (or install) to display a stupid "Hi, {name}.". The WordPress team is actively trying to solve this problem, with the Block Bindings API being a recent major step forward. However, the functionality is arguably not yet mature enough for complex commercial use.

A few issues spring up from the increased importance of the database. It used to be the case that the database was meant to store the data and the theme was responsible for displaying that data. This is not the case anymore, as Full Site Editing means the templates are stored in the database.

This proves problematic, especially in professional settings. How do you version control a database? It's not like you can use it with git to track your changes. You're stuck with the built-in revision system. An even better question is: how do you push your changes from staging to production?

You can copy the modified markup and manually paste it in the code editor on the production site, but that's brittle and annoying. You can export your theme every time you make a change and install it on production, but then you're overriding the base theme, and you better pray your user(s) don't make any changes in the Site Editor. You can try syncing the database, maybe even partially, but that's dangerous and creates a whole new set of data syncing problems.

Last but not least, you have to remember that block themes and FSE are still under heavy development. The ecosystem is less mature than that of classic themes. Not all plugins are compatible. Major architectural changes can happen with every update, while the best practices are constantly being redefined. It's an immature ecosystem, not fit for a project requiring stability.

Hybrid Themes

A hybrid theme is a classic theme that has adopted some block theme functionality. This is not an official term, but every professional WordPress developer will understand you if you call a theme "hybrid".

Hybrid themes exist for a couple of reasons. Some classic themes predating the block system are so complex and widely used it'd be impossible to convert them to block themes. Yet, the block-based APIs have been evolving over the past few years, and it's very clear blocks are not going away any time soon.

It's in some limited ways possible for classic themes to utilize block theme functionality without becoming a block theme. Let me remind you - a block theme is a theme with a /templates/index.html file present.

There's no single set of things a classic theme has to do or support in order to be classified as a hybrid theme. The most relevant things to look for are either add_theme_support() calls enabling block-based functionalities or the presence of a theme.json file, with emphasis on the latter. Yes, a classic theme can have a theme.json file.

You can use the theme.json to configure settings and styles, just like you can in block themes. This is, in it of itself, very powerful. You can provide color palettes and default styling to blocks, which is invaluable even in classic themes, as the Block Editor is now the default editor for every post and page.

Keep in mind though, the Site Editor experience will be very limited. I have not found a way to enable style editing (only previewing) or to even get the navigation, pages, or templates tabs to show up. The latter is to be expected as hybrid themes are not meant to support FSE - they are still classic themes after all.

You can also create HTML template parts and use them in your PHP templates. You can render a block-based part in PHP using the block_template_part() function. You can even edit template parts and add patterns in the patterns tab of the Site Editor, just like you could in block themes.

If you add HTML templates to your theme's /templates directory, they'll be used instead of their PHP counterparts. Yes, block templates can be used and are prioritized even if the theme is not a block theme. But as already noted, you can forget about editing them from the Site Editor.

PS: The Site Editor in hybrid themes will most likely be named "Design" instead of "Editor" in the Appearance menu.

Plugins

We have already covered the first 2 out of 3 major layers of WordPress - content and presentation. It's time for the last one - functionality.

You add functionality to WordPress websites with plugins. This decouples the presentation layer (themes) from the functionality layer. Some of the things we've already discussed that you should do in plugins are:

- Adding custom post types

- Adding shortcodes
- Adding blocks
- Adding widgets

Plugins are only possible because of WordPress's event driven architecture, powered by its hooks system. If you're feeling a bit rusty in hooks, this might be a good time to revisit that chapter.

At the very lowest level, a plugin is just one PHP file with a comment header, placed in the /wp-content/plugins directory. This tiny system, coupled with access to all of WordPress's hooks, unlocks virtually limitless possibilities.

To demonstrate plugins, we'll create a very simple Post Reading Time plugin, displaying the estimated reading time of a post. This will give us an opportunity to demonstrate every major concept while being simple enough to not clutter the document with too much code. We will continue extending this plugin with additional functionality throughout subsequent parts of this guide.

The official and up-to-date documentation of plugins is available in the Plugin Handbook.

Best Practices

There are a few best practices you should keep in mind when developing a plugin.

Avoid Naming Collisions

That shouldn't be a surprise to you. We've already been doing that with the thm_ prefix. You should prefix all of your global-facing touchpoints with a unique slug. That includes, but is not limited to:

- functions
- classes
- variables
- options
- custom database tables
- CSS classes

Alternatively, you may consider using unique PHP namespaces to hide them from the global namespace. Another viable and popular approach is to declare all of your methods in a class instead of as functions. The benefit is that you only have to ensure the class's name is unique, and you don't have to worry about its methods.

Whatever you do, just keep in mind that your plugin has to be interoperable with the thousands of other plugins available in the WordPress ecosystem. We will use a pgn_ prefix (although it's recommended for your prefix to be at least 5 characters long).

Check For ABSPATH

'ABSPATH' is a global constant defined by WordPress during its boot process. It's the de-facto way of checking if the file has been loaded by WordPress or if it's been accessed directly. You should check it in all of your PHP files.

```
if ( ! defined( 'ABSPATH' ) ) {
    exit;
}
```

If you don't do that, someone can access your script by navigating directly to https://domain.com/wp-content/my-plugin/insecure-file.php. The code will be run without the WordPress core ever being loaded.

Best case scenario - the user gets a 500 status code because the used wp functions aren't defined. Worst case scenario - the code messes with the database or files on your server, corrupting your website or introducing vulnerabilities. Get in the habit of checking for ABSPATH.

Don't Reinvent The Wheel

This should go without saying, but remember to always do things the WordPress way (unless you have a really good reason not too). One of the biggest problems with plugins, and why it's so hard to maintain websites with too many of them, is incompatibility.

Use core APIs whenever you can. Don't try to go around what everybody else is doing and don't try to fight the framework. You're risking incompatibilities with other plugins and with WordPress itself if it ever introduces breaking changes. The core APIs will surely be adapted - your code will not.

Plugin Architecture

It's important to understand that your plugin is basically a full-fledged backend application. You have complete control over everything that happens. You have no template hierarchy that you have to abide by. Your single main file is loaded by WordPress, and that's the front controller to your code.

You can create as many or as few files as you want. You can use whatever coding practices you want, be it procedural or object-oriented. Hell, you can even use the MVC pattern in your plugins. No one is going to stop you.

The Main File

A plugin can consist of only one file in the /wp-contents/plugins directory. You don't even need a folder for this file. WordPress parses all the files in this directory (and subdirectories) and looks

for specific header comments. A file with such comments becomes the main plugin file. Here is an example:

```
/*
 * Plugin Name:       My Basics Plugin
 * Plugin URI:        https://example.com/plugins/the-basics/
 * Description:       Handle the basics with this plugin.
 * Version:           1.10.3
 * Requires at least: 5.2
 * Requires PHP:      7.2
 * Author:            John Smith
 * Author URI:        https://author.example.com/
 * License:           GPL v2 or later
 * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
 * Update URI:        https://example.com/my-plugin/
 * Text Domain:       my-basics-plugin
 * Domain Path:       /languages
 * Requires Plugins:  my-plugin, yet-another-plugin
 */
```

I copied this from the official documentation. I'm not going to explain every single attribute. You can look them up any time. The only required field is 'Plugin Name'. If that is present, the plugin will be recognized and displayed in the admin panel. You should only have one main file for your plugin.

Code & File Structure

As I already explained, you can structure your plugin whatever way you want. This is not a programming tutorial so I'm not going to teach you the best way to write readable code. What I will show you are some commonly seen patterns among plugin developers.

Single File

This is the simplest structure ever. You only have your main file. This single file contains all of your code: functions, classes, hooks, etc. It is very simple and readable, until it's not. If your plugin has over 200 lines of code, it's usually a sign you should use a different structure.

This is what we are going to be using for our Post Reading Time plugin. We will also not use OOP, just functions. That's because it's the simplest and least verbose, which is important for being able to show it in this guide.

Multiple Files - Only Functions

This is a step in the right direction. It ensures a better structure of the project by splitting different parts into different files. The potential problem is that it still operates in the C-style,

procedural paradigm, with functions and file imports. You can do that if you like, but the object-oriented paradigm is usually preferred.

A potential file structure might look like this:

- **my-plugin/**
    - *my-plugin.php* (main file)
    - **includes/**
        - *functions.php*
        - *admin.php*
        - *shortcodes.php*
    - **assets/**
        - **css/**
        - **js/**
    - **languages/**

Single Class

This can either be all in the main file or with a separate class file. The defining factor is the use of a single class, usually following the Singleton pattern. You might use that as a technique to avoid naming collisions of your methods.

Some functionalities can naturally be modeled using a single class, or it might be an intermediary state that will change as the scope of the plugin grows. Either way, OOP is usually a step in the right direction.

Multiple Classes

This is where most commercial plugins live. It's just a typical OOP application with classes and objects interacting with each other. You might have separate classes for handling the admin and frontend sides.

A potential file structure might look like this:

- **my-plugin/**
    - *my-plugin.php* (main file)
    - **includes/**
        - *class-my-plugin.php*
        - *class-my-plugin-admin.php*
        - *class-my-plugin-public.php*
    - **assets/**
        - **css/**
        - **js/**
    - **languages/**

And then you might see something like this somewhere in the code:

```
if ( is_admin() ) {
      $admin = new My_Plugin_Admin();
      $admin->run();
} else {
      $public = new My_Plugin_Public();
      $public->run();
}
```

Post Reading Time Plugin (Code Example)

Let's write our plugin. Here's how it'll work. We'll hook into the the_content filter which runs on every the_content() call and passes the $content as an argument. We'll strip the HTML tags, calculate the number of words, and divide by 250 which is the average reading speed. We'll then display our estimated rounded up reading time before the actual content. Here's the entire code:

```
<?php
/*
* Plugin Name: Post Reading Time
*/

if ( ! defined( 'ABSPATH' ) ) {
    exit;
}

function pgn_calculate_reading_time( $content ) {
    $avg_words_per_minute = 250;
    $word_count = str_word_count( strip_tags( $content ) );
    $reading_time = $word_count / $avg_words_per_minute;

    return ceil( $reading_time );
}

function pgn_add_snippet_to_post_content( $content ) {
   if ( ! is_single() || ! is_main_query() || ! in_the_loop() ) {
        return $content;
    }

    $reading_time = pgn_calculate_reading_time( $content );
    $display_html = '<div class="pgn-reading-time">Reading time: ' . $reading_time .
' min</div>';
```

```
    return $display_html . $content;
}
add_filter( 'the_content', 'pgn_add_snippet_to_post_content' );
```

That's it. The plugin is done. You can publish it on the Plugin Directory and add "WordPress Plugin Developer" to your LinkedIn profile.

Loading Assets

What if you wanted to load a CSS file to style your output? Or a JS file to spice it up on the frontend? Well, it turns out it's not much different than it was with themes.

We're still loading assets using the enqueue system. The only real difference is how we get the URL and path to the file. Let's add a style.css file to our plugin directory and load it on the frontend. Here's the additional code we need to add to our main file:

```
function pgn_enqueue_styles() {
    wp_enqueue_style(
        'pgn-style',
        plugins_url( 'style.css', __FILE__ ),
        [],
        filemtime( plugin_dir_path( __FILE__ ) . 'style.css' ) // auto-versioning
trick
    );
}
add_action( 'wp_enqueue_scripts', 'pgn_enqueue_styles' );
```

Remember the functions for getting the file's URL or path in themes? Like get_theme_file_uri(), get_template_directory(), etc. There are 4 of those for plugins:

- **plugins_url($path, $plugin)** - retrieves a URL within the plugins directory.
    - **$path** - path appended to the end of the URL.
    - **$plugin** - a full path to a file inside the plugin directory. The URL will be relative to its directory. If not supplied, the URL will use the plugins directory (usually /wp-content/plugins/).
    - The most common and robust way to use plugins_url() is by providing two arguments: the relative path to the asset, and the magic constant __FILE__. plugins_url('style.css',__FILE__) tells WordPress: "Return the URL for style.css located in the same directory as the current PHP file.".
- **plugin_dir_url($file)** - wrapper over plugins_url('', $file). Returns the URL directory path for the file passed (with trailing slash). E.g., https://example.com/wp-content/plugins/post-reading-time/.

- **plugin_dir_path($file)** - same as plugin_dir_url() but returns the filesystem path rather than a URL. It's a wrapper for trailingslashit(dirname($file)).
- **plugin_basename($file)** - returns the relative path to the plugin file in the plugins directory. E.g., 'post-reading-time/includes/passed-file.php'.

Plugin Settings

Every sufficiently large plugin needs to provide the user with some level of control. For demonstration purposes, we'll create a settings page for our plugin allowing the user to specify a different string for the "Reading time" text.

Options API

The Options API is the layer between you and the wp_options database table. All settings should be stored in that table. As a matter of fact, we've already used this API indirectly when we covered the Customizer. Our settings were stored in the options table and the Customizer API uses the Options API to manage them.

The API is stupidly simple. It consists of only 4 main functions providing the CRUD functionality:

- add_option($option, $value)
- update_option($option, $value)
- get_option($option, $default_value)
- delete_option($option)

In reality, you'd usually use update_option() instead of add_option(), as it creates the option automatically if it doesn't yet exist (just like update_post_meta() does). Values can be either single values or arrays, which will be serialized. If you have many options, it's usually better to store them as an array, as that reduces the number of rows in the database.

There's also another parameter you can pass to the add and update functions that I didn't include - $autoload. It's a boolean parameter. If it's true, the option will be preloaded by WordPress while it's booting up. It will then be cached in memory.

Why? Imagine you had 15 different options. If you didn't autoload them, every get_option() call would make a separate SELECT query. By the time your plugin was done, you would've made 15 database queries. Now multiply that by 20 plugins installed on the website. With autoloading, WordPress makes only one query to fetch all autoloaded options. All subsequent get_option() calls for any of these options are served from in-memory cache.

It's a balancing game, as the option will be loaded on every page, even if it isn't used - slowing down the entire website. You should use autoload if your code fetching the options runs on many requests, especially on the frontend. If your options are only used on a few rarely visited URLs, autoloading will probably do more harm than good.

PS: If you visit the /wp-admin/options.php page, you can see all of your options stored in the wp_options table.

Settings API

The Settings API is a standard WordPress way of creating the frontend in wp-admin for modifying your options. It lets you create settings, fields for those settings, and sections for those fields. Every section gets rendered on a settings page, which is covered in "Administration Menus".

There are 3 main functions for creating settings (arguments not shown):

- **register_setting()** - registers a new option with the supplied name.
- **add_settings_field()** - registers a field to be displayed. A callback is responsible for outputting the HTML of the field (i.e., an <input> element). The input's 'name' attribute must match the name of the option registered with register_setting(). The field gets connected with a specified section.
- **add_settings_section()** - registers a settings section. The section gets connected with a specified page.

Similarly, there are 3 main functions for displaying settings (arguments not shown):

- **settings_fields()** - renders required hidden utility and security fields.
- **do_settings_fields()** - renders settings fields for a given section (not usually used directly).
- **do_settings_sections()** - renders all settings sections connected with the specified page. Uses do_settings_fields() internally.

It's important to know that you don't have to use the Settings API. You can just as well create the entire page yourself. There are, however, significant benefits from using the API:

- **Robustness** - think back to the list of best practices. You want to use the built-in API. Not only is it better tested and safer; it will also change as WordPress changes, and your code will not.
- **Visual consistency** - your fields and other elements will be styled the same way all of the other core elements are. This improves visual consistency throughout the entire admin panel.
- **Less work** - if you were to create a custom settings page, you'd have to handle form submissions and implement all of the security measures that come with that. Not to mention the fact that you would just have to write a lot more code.

I know this is a pretty heavy theory, and it's fine if you don't fully understand it yet. Hopefully it'll all click once you see the code example.

PS: register_setting() doesn't provide any way of enabling or disabling autoload for the option. It gets registered with a value of 'auto'. If you need to deliberately set autoload, you would have to add the option yourself using add_option(), usually during plugin activation. Even if you do that,

you'd still have to register it as a setting using register_setting(), but it won't add the option to the database since it already exists.

PS 2: You can also display errors to users with functions like add_settings_error(), but I deemed it not important enough to include in this section. Go read the documentation.

Administration Menus

Administration menus are the panels on the left side of the admin dashboard. There are 2 types of them - top-level menus and sub-menus.

Top-level menus are the ones you always see. These are 'Posts', 'Pages', 'Appearance, Plugins, etc. To add a top-level menu, you have to call the add_menu_page() function. You need to supply a callback responsible for rendering the page, usually with some boilerplate HTML and by calling methods like settings_fields() and do_settings_sections().

Sub-menus are menus nested under top-level menus. They only show up when the user is in the top-level menu or hovers over it. To add a sub-menu, you have to call the add_submenu_page() function. You need to specify the parent top-level menu and, once again, create a callback responsible for rendering the page's HTML.

There is an exhaustive list of helper functions you can use instead of add_submenu_page() to register a sub-menu for the built-in top-level menus. These are just simple wrappers over the core function with a hard-coded top-level menu identifier. Here they are:

- add_dashboard_page()
- add_posts_page()
- add_media_page()
- add_pages_page()
- add_comments_page()
- add_theme_page()
- add_plugins_page()
- add_users_page()
- add_management_page()
- add_options_page()

The general rule is to not create a top-level menu unless your plugin is big and actually requires multiple sub-menus. That's to not clutter the UI. I don't know if I fully agree with that. Sometimes it's a better user experience to see the menu in the sidebar than having to scour the entire admin panel to find it. I'm probably wrong though, so do as you see fit.

On a side note, you don't actually have to add a menu page to display your fields. You can connect your section to a page that already exists, or even connect your fields to a section that already exists. This is, in my opinion, terrible UX. I'd personally almost never hide my settings in between other core settings.

Settings Page (Code Example)

Let's start with registering the settings:

```
// Optional callback function executed at the top of the section
function pgn_text_section_html() {
    echo '<p>Settings related to text</p>';
}

// Callback function rendering the field's input element
function pgn_reading_time_text_html() {
    $current = get_option( 'pgn_reading_time_text' ); // We have to display the
current value
    echo '<input type="text" name="pgn_reading_time_text" value="' . $current . '">';
}

function pgn_settings_init() {
    // First is $option_group, second is $option_name (the one stored in the database)
    register_setting( 'pgn_post_reading_time_group', 'pgn_reading_time_text' );

    add_settings_section(
        'pgn_text_section',         // $id - unique ID of the section
        'Text Settings',            // $title - displayed as heading
        'pgn_text_section_html',    // $callback
        'pgn_post_reading_time_menu'// $page - slug of our menu page
    );

    add_settings_field(
        'pgn_reading_time_text',                // $id
        'Text alternative to "Reading time"',   // $title - label
        'pgn_reading_time_text_html',           // $callback
        'pgn_post_reading_time_menu',           // $page
        'pgn_text_section'                      // $section
    );
}
add_action( 'admin_init', 'pgn_settings_init' );
```

Before we create the sub-menu, let's stop and discuss what's going on here. The $option_group parameter passed to register_setting() is a particularly interesting thing. It's like an ID for a group of settings. Every setting has to be in a group, and all of the settings displayed on one page are supposed to be in the same group.

As far as I know, it's mostly used as a security mechanism. $option_group is a hidden field included in the form element. WordPress checks its value and only updates options sent in the POST request if they have been registered for this group. This prevents options from being updated by forms that aren't supposed to update them.

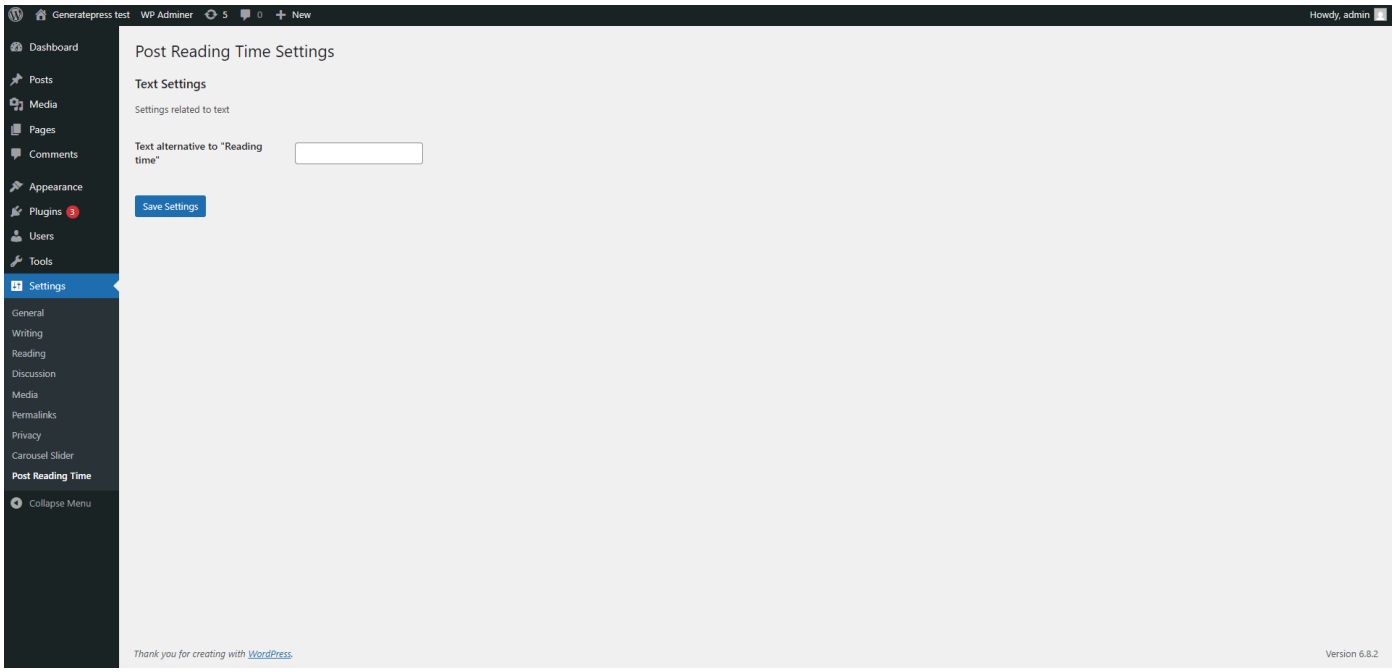
Other than that, you can see that we register the setting, add its field, connect it with a section, and connect the section (and the field) with a page. A page we are going to add right now:

```
// Callback function rendering the contents of the page
function pgn_settings_page_html() {
    ?>
    <div class="wrap">
        <h1><?php echo get_admin_page_title(); ?></h1>
        <form action="options.php" method="post">
            <?php
            settings_fields( 'pgn_post_reading_time_group' );
            do_settings_sections( 'pgn_post_reading_time_menu' );
            submit_button();
            ?>
        </form>
    </div>
    <?php
}

function pgn_settings_page_init() {
    add_options_page(
        'Post Reading Time Settings', // $page_title - displayed in <title>
        'Post Reading Time',          // $menu_title
        'manage_options',             // $capability - don't worry about that now
        'pgn_post_reading_time_menu', // $menu_slug - slug & unique ID
        'pgn_settings_page_html'      // $callback
    );
}
add_action( 'admin_menu', 'pgn_settings_page_init' );
```

As you can see, we only have to write a very tiny amount of boilerplate HTML. That's because the Settings API is doing the heavy lifting. The docs recommend you add the div with a wrap class, but they don't explain why.

The action of the form is set to options.php. This is the core file that handles the form submit and option updates. It's where the magic of the API happens. settings_fields() renders the hidden fields, including the aforementioned field with the 'pgn_post_reading_time_group' option group name.

You can see we registered the sub-menu using the helper function add_options_page(). I just think it's more self-documenting and makes the code prettier. Here is what the final page looks like:



Updated Plugin (Code Example)

We have the option in the wp_options table. We have created a page for the user to modify it. It's time to use it in the code. Here's our new snippet function:

```
function pgn_add_snippet_to_post_content( $content ) {
    if ( ! is_single() || ! is_main_query() || ! in_the_loop() ) {
        return $content;
    }

    $reading_time_text = get_option( 'pgn_reading_time_text' );
    if ( empty( $reading_time_text ) ) {
        $reading_time_text = 'Reading time'; // Default
    }

    $reading_time = pgn_calculate_reading_time( $content );
    $display_html = '<div class="pgn-reading-time">' . $reading_time_text . ': ' .
$reading_time . ' min</div>';

    return $display_html . $content;
```

```
}
```

Now your users can freely change the text displayed on their websites. Success.

Activation, Deactivation, Uninstall

There are 3 main events in the lifecycle of a plugin that you can hook into:

- Activation
- Deactivation
- Uninstall

Activation happens when the user clicks "Activate" after installing your plugin. You can hook a callback function to this event using the register_activation_hook() function. Some of the most common tasks to do here are:

- Adding options (if you need to control autoload or are not using the Settings API).
- Creating database tables (if you're using custom tables).
- Flushing rewrite rules (if you're registering custom post types or anything like that).
- Registering custom cron jobs.
- Redirecting to the welcome/tutorial page.

Deactivation happens when the user clicks "Deactivate". You can hook a callback function to this event using the register_deactivation_hook() function. It's important to note that no data should be deleted on the deactivation hook. The user is temporarily turning the plugin off and expects all of their settings to still be there when they turn it back on. Some things to do here are:

- Flushing rewrite rules (similarly as during activation, flush if your plugin changes them).
- Removing custom cron jobs.
- Flushing cache/temp.

Uninstall happens when the user clicks "Delete" after they deactivate the plugin. In contrast to deactivation, this event is supposed to be destructive. The user is getting rid of your plugin. You should delete all data your plugin has added, either in the database or in the filesystem. A good plugin leaves no trace of ever being installed.

There are 2 ways of running the uninstall code: you can either register a callback with register_uninstall_hook() or you can create an uninstall.php file in your plugin's root directory. The latter is a recommended best practice, as it is loaded in a sandboxed environment and keeps the destructive logic separated from your main plugin files.

Instead of checking for ABSPATH, you should check for WP_UNINSTALL_PLUGIN. Let's add the uninstall.php to our plugin to delete the reading time text option.

```php
<?php
if ( ! defined( 'WP_UNINSTALL_PLUGIN' ) ) {
    exit;
}

delete_option( 'pgn_reading_time_text' );
```

Hooks In Plugins

Just like the core WordPress files, the user shouldn't modify plugin files. Their changes will be overridden on the next plugin update. This is the reason why hooks exist in the first place, and plugins are no different.

The hallmark of a great plugin is in its hooks. You should always try to predict what data your users might want to filter or where they might want to run actions. The most important and commonly used settings should be exposed in the UI (e.g., using the Settings API), but some requirements are just so niche that displaying them in the admin panel would only clutter it. Nevertheless, that doesn't make them any less valid.

Trust me when I say that you're going to lose your mind if you ever have to modify a plugin's behavior, only to discover that there's no hook to actually do it. With that in mind, let's make our tech-minded users love us by making our plugin extensible.

```php
function pgn_calculate_reading_time( $content ) {
    $avg_words_per_minute = 250;
    $avg_words_per_minute = apply_filters( 'pgn_words_per_minute',
$avg_words_per_minute );

    $word_count = str_word_count( strip_tags( $content ) );
    $reading_time = $word_count / $avg_words_per_minute;
    $reading_time = apply_filters(
        'pgn_reading_time',
        $reading_time,
        $word_count,
        $avg_words_per_minute
    );

    return ceil( $reading_time );
}

function pgn_add_snippet_to_post_content( $content ) {
    if ( ! is_single() || ! is_main_query() || ! in_the_loop() ) {
```

```
        return $content;
    }

    $reading_time_text = get_option( 'pgn_reading_time_text' );
    if ( empty( $reading_time_text ) ) {
        $reading_time_text = 'Reading time'; // Default
    }

    $reading_time = pgn_calculate_reading_time( $content );
    $display_html = '<div class="pgn-reading-time">' . $reading_time_text . ': '
. $reading_time . ' min</div>';
    $display_html = apply_filters( 'pgn_display_html', $display_html,
$reading_time_text, $reading_time );

    return $display_html . $content;
}
```

As you can see, we've added 3 filters:

- pgn_words_per_minute (which should probably also be a setting. "Also" and not "instead" because filters provide programmatic control, i.e., returning a different value depending on the post)
- pgn_reading_time
- pgn_display_html

Our users (or any extending plugins) can now hook into those filters with add_filter() and modify the data. We didn't add any actions because, in this particular case, I don't see any legitimate reason/place to do so.

Please note that, similar to other code snippets in this guide, the above code is not production ready. You should always check that the values returned by the filters are what you expect them to be. For example - for words per minute, you'd check that the value is a positive integer, and if it isn't - you'd default to 250 and throw an error (using wp_trigger_error() or a custom function checking for 'WP_DEBUG' if you want to support <6.4).

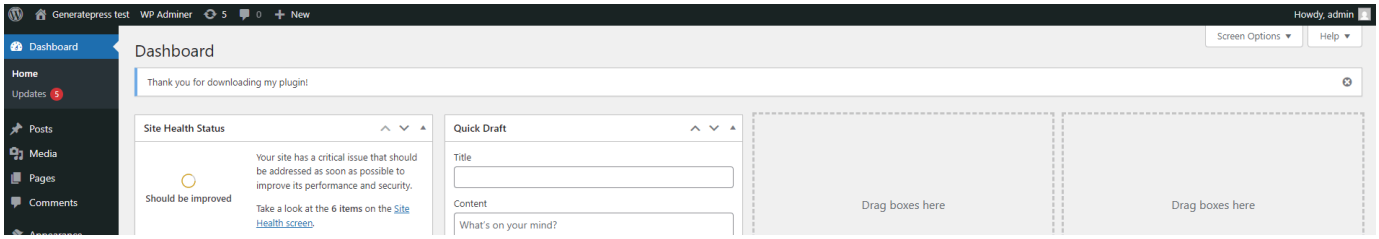
Admin Notices

Admin notices are the standard WordPress way of communicating with the user. The system is incredibly simple - you add an action rendering the HTML on the 'admin_notices' hook. Let's say you're creating an incredibly annoying plugin, which displays a "Thank you for downloading my plugin!" notice on every page. Here's the code:

```
function pgn_annoying_welcome_message() {
```

```
    echo '<div class="notice notice-info is-dismissible"><p>Thank you for
downloading my plugin!</p></div>';
}
add_action( 'admin_notices', 'pgn_annoying_welcome_message' );
```

And here's what you'll see at the top of every page of the admin panel:



A wrapper div with the 'notice' class is required for the message to be styled. The 'notice-info' class follows the 'notice-{type}' pattern to set the type of the notice. There are 4 types of notices, each of which uses a different accent color:

- **info** - blue
- **warning** - yellow/orange
- **error** - red
- **success** - green

Don't get fooled by the 'is-dismissible' class. It's responsible for adding the "x" on the right to close the notice, but it will not persist when you refresh the page. It is the plugin author's responsibility to render notices only when appropriate and provide a way to dismiss them. This usually means conditionals in the hooked function and/or custom admin JavaScript (AJAX) communicating with the backend to save the dismissed state in the database.

Must-Use Plugins

Must-use plugins are a pretty niche topic. These are PHP files placed in the /wp-content/mu-plugins/ directory. Some characteristics and differences from normal plugins are:

- They are loaded before normal plugins (in alphabetical order).
- They don't require any header comments.
- They are not displayed on the plugins page (only in a separate 'Must-Use' tab).
- They can't be added via the admin panel (only manually, e.g., using FTP or SSH).
- They can't be updated via the admin panel.
- They can't be disabled in the admin panel (you'd have to delete the file to disable it).
- WordPress only looks for them in the /wp-content/mu-plugins/ directory, not in subdirectories.

Great, but why? Why do must-use plugins even exist? mu-plugins are great at ensuring that functionality critical to the site is never disabled. Imagine you're making a website for a client and the website requires a custom post type. If you added it in a normal plugin and the website owner disabled it by accident, the entire site would crash. That's a good reason to make it a must-use plugin.

Some plugins and themes also add their own mu-plugins. This is a rather controversial topic. Many of them have good reasons, often stemming from the fact that must-use plugins are loaded before normal plugins. That being said, it can be perceived as a breach of contract. When you install a normal plugin, you expect it to be a normal plugin, and not add persistent code that you can't even delete from the dashboard.

Note that, because WordPress doesn't search for mu-plugins in subdirectories, it's often the best practice to make the actual PHP file just a simple loader responsible for requiring appropriate files. These can be files in a subdirectory of the mu-plugins folder, or more commonly with public plugins, files inside the /wp-content/plugins/the-plugin-folder/ directory.

Do you see how the previous example introduces persistence? The files loaded by the plugin's mu-plugin code will be loaded even if the plugin is deactivated. It's therefore the plugin's responsibility to manage its must-use plugin, potentially removing it on the deactivation hook and updating it if needed.

Because most website owners and developers do not even know that mu-plugins exist, they are an attractive attack vector for hackers. Hackers have been known for putting malicious code in the mu-plugins directory to achieve persistence. Most people, even if they know they've been hacked, don't know to check this directory for installed plugins, making them oblivious to the running malicious code.

Internationalization & Localization (Translations)

It'd be a shame if you created a great theme or developed an amazing plugin but only English speakers could use it. Internationalization (i18n) and localization (l10n) are required for your strings to be translated into multiple languages.

Let's start by clearing up those 2 terms:

- **i18n** is the practice of making your code translatable.
- **l10n** is the practice of translating the code. You (or more likely a translator) are manually translating every hard-coded string.

WordPress uses the GNU gettext system. As such, there are many commonalities. Translatable strings have to be wrapped in a special internationalization function. The code is then parsed and a .pot (Portable Object Template) file is generated. This file is the template to .po (Portable Object) files, which contain the final translations. These .po files are then compiled into optimized .mo (Machine Object) files, which are used by WordPress to serve translated strings.

Don't worry if you don't get it yet. You will by the end of this chapter. Let's get into the details.

i18n In PHP

There are a few essential internationalization functions you have to know:

- __()
- _e()
- _n()
- _x()

__($text, $domain) is the fundamental "return translated string" function. Here's how to use it:

```
$translated = __( 'String to translate', 'post-reading-time' );
```

The first parameter is the string to be translated, that's obvious. But what's the second one? Well, that's the text domain. The text domain ensures the translations of your theme or plugin don't get mixed with another theme or plugin. How else would WordPress know which translation file to use if two plugins contained the same string? It's the unique identifier for all your internationalized strings.

The text domain has to always be passed as a literal string. It can't be a variable. That's because this code is later statically parsed to generate the translation file.

_e($text, $domain) is an alias to "echo __($text, domain)". It echoes the translated string instead of returning it.

_x($text, $context, $domain) is a translation function used when you need to add context to the string. An example of when that's useful is for words which can serve both as a verb or a noun. The second argument becomes the context, e.g.:

```
$translated = _x( 'Post', 'noun', 'post-reading-time' );
```

This will create a separate translation entry from the word "Post" with the context "verb". This way, the translator can translate them separately, as most languages will have different words for them. There's also an _ex() version of this function. It echoes the string instead of returning it.

_n($single, $plural, $number, $domain) is for strings containing variable numbers. Different languages have different rules when it comes to singular and plural numbers. Just think about "1 post, 2 posts". These rules get way more messy for different languages, and we'll get to that in a moment. Here's a usage example:

```
$translated = _n( '%s comment', '%s comments', $count, 'post-reading-time' );
```

If $count is equal to 1, the first string will be returned. If it's greater than 1, the second one will be returned. Note that the %s placeholder will not be replaced with $count. It's the developer's responsibility to do that later (probably using printf).

Some niche cases require context for strings containing numbers. Thankfully, there's a function for that:
_nx($single, $plural, $number, $context, $domain). This works just like _x(). Its most important utility is for disambiguating homonyms:

```
$soccer_matches = _nx( '%s match', '%s matches', $n, 'soccer games',
'post-reading-time' );
$wooden_matches = _nx( '%s match', '%s matches', $m, 'matchsticks',
'post-reading-time' );
```

Unfortunately, there are no _en() or _exn(), so you'll have to echo the outputs yourself.

Dates & Numbers

There are 3 special situational functions:

- date_i18n()
- wp_date()
- number_format_i18n()

date_i18n() is an obsolete function used to internationalize a date in a given format. Its job is translating strings like the weekday and month names. This function is a little confusing so please read its documentation if you ever have to use it.

Why did I say it was obsolete? Because it got superseded by **wp_date()** in WordPress 5.3. The old function was quirky and problematic, namely - you had to take care of offsetting the timestamp to match your timezone.

date_i18n() does not care about your site's timezone. It only translates whatever date you give it. If your site's timezone was Europe/Warsaw, you'd have to manually offset the timestamp by 1 hour for winter time (UTC+1), and by 2 hours for summer time (UTC+2). As you might imagine - that's brittle.

wp_date() takes care of that for you. It expects a UTC timestamp and a separate timezone argument (defaults to the site-wide timezone specified in settings). It's DST aware (winter vs summer time) and just as date_i18n(), it takes care of string translations. You should always use wp_date() when dealing with dates in WordPress.

number_format_i18n(float $number, int $decimals) formats a float number based on the locale. It returns a string. Different regions have different standards when it comes to numbers. Some use periods, some use commas, and some use spaces. Here's an example:

```
$num = number_format_i18n( 1254.5, 2 );
```

$num will equal "1,254.50" for US English and "1 254,50" for Polish.

Variables

What do you do if you have a variable in the string, like "Your city is $city"? You can't just use this string inside an i18n function. If you did that, the translator would translate its static version ("Your city is $city"). But then during the code's execution, it'd become ""Your city is London", which would have no matching translations.

The correct way is to use printf (or sprintf), usually with a helpful comment:

```
printf(
      /* translators: %s: Name of a city */
      __( 'Your city is %s.', 'my-plugin' ),
      $city
);
```

The string to be translated is just a static string with a %s placeholder. The comment is optional, but it can be a helpful hint for translators. Otherwise, they won't know what the final sentence will be. Is your city "Paris" or is it "crowded"?

Note that the translator can do anything with this string, as long as the translated version contains the %s. For example, they could place it at the beginning of the sentence, which may be the logical flow for some languages.

A string with multiple variables might look like this:

```
printf(
      /* translators: 1: Name of a city 2: ZIP code */
      __( 'Your city is %1$s, and your zip code is %2$s.', 'my-plugin' ),
      $city,
      $zipcode
);
```

.pot, .po, And .mo Files

Great, you know the most important functions for internationalizing your code, but how does it *really* work? Isn't that what this guide is for?! Yes it is, and you're going to learn it, but we have to start with the mysterious translation files.

Note that we're now shifting from internationalization (making code translatable) into localization (translating).

POT Files

The Portable Object Template file is the template you're going to give to translators. It's generated automatically. There are many tools to do that, but the WordPress way is to use WP-CLI "wp i18n make-pot" command. This command parses all of the files in your plugin or theme directory and generates a POT file with all the strings wrapped in i18n functions.

Here's the post-reading-time.pot file generated by this command now, when our plugin is not yet internationalized:

```
# Copyright (C) 2025
# This file is distributed under the same license as the Post Reading Time
plugin.
msgid ""
msgstr ""
"Project-Id-Version: Post Reading Time\n"
"Report-Msgid-Bugs-To:
https://wordpress.org/support/plugin/post-reading-time\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"POT-Creation-Date: 2025-08-27T12:18:32+00:00\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"X-Generator: WP-CLI 2.12.0\n"
"X-Domain: post-reading-time\n"

#. Plugin Name of the plugin
#: post-reading-time.php
msgid "Post Reading Time"
msgstr ""
```

PO Files

Portable Object files are just the filled out templates. The translator gets a POT file, translates all the strings, and saves it as a .po file. Here's what the translation of our current 'Post Reading Time' plugin looks like (only the relevant part):

```
#. Plugin Name of the plugin
#: post-reading-time.php
msgid "Post Reading Time"
msgstr "Czas Czytania Wpisu"
```

This file would then be saved as post-reading-time-pl_PL.po. The filename format is {text-domain}-{locale}.po.

MO Files

The MO files are compiled PO files. These use a binary format which makes them more efficient. Once again, you can compile your .po files into .mo files using different tools. The most fundamental one is the gettext's msgfmt command. For us, that'd be:

```
msgfmt -o post-reading-time-pl_PL.mo post-reading-time-pl_PL.po
```

The PO filenames really didn't *have* to abide by the {text-domain}-{locale}.po format, but the MO files do because that's how they are found and loaded by WordPress.

Loading The MO Files

You've internationalized your code, you translated it, and you generated the .mo files. That's great, but WordPress has no idea about them until you load them. There are a few ways of achieving that, and a couple of nuances to keep in mind.

PS: This process is very different for themes and plugins hosted on wordpress.org. There's an entire section on that later on.

load_plugin_textdomain() & load_theme_textdomain()

Let's assume we created a /languages subdirectory for our plugin. The path to that directory would be /wp-content/plugins/post-reading-time/languages. This folder would contain all our .mo files. The first way of loading those translations would be explicitly calling load_plugin_textdomain(). Here's an example:

```
function pgn_load_translations() {
    load_plugin_textdomain(
```

```
        'post-reading-time',
        false,
        dirname( plugin_basename( __FILE__ ) ) . '/languages'
    );
}
add_action( 'init', 'pgn_load_translations' );
```

The first argument is the text domain, the second one is deprecated (just make it false), and the third one is the path relative to the /wp-content/plugins/ directory. WordPress will then load all the MO files in this directory with the matching text domain in the filename.

By the way, there's also load_theme_textdomain() and load_child_theme_textdomain().

Domain Path

So that was your first option - explicitly calling a PHP function. The second option is with a 'Domain Path' header comment. You have to place this comment either in styles.css (for themes) or in the plugin's main file (for plugins). The value of that parameter is the relative path to the directory containing MO files. Example:

```
/*
* Plugin Name: Post Reading Time
* Domain Path: /languages
*/
```

WordPress will automatically find and load all your MO files in this folder. No explicit function call is needed.

I've mentioned that there are a couple of nuances to keep in mind. Here's the first one. load_plugin_textdomain() and 'Domain Path' are not really the same. If you only load your MO files explicitly without specifying the 'Domain Path', then your plugin/theme metadata won't be translated.

Think about it. We translated our plugin's name from English "Post Reading Time" to Polish "Czas Czytania Wpisu". What happens when the user disables the plugin? Our code won't run and the translations won't be loaded, yet the name will still be displayed in the admin dashboard.

If you specify 'Domain Path', the MO files will always be loaded, even if the plugin is disabled. That's why it's usually better to use 'Domain Path' instead of load_plugin_textdomain().

Text Domain

There's one more header comment relevant to translations and it's a source of more complex nuances. The header in question is 'Text Domain'. This property, as you might have guessed, specifies the text domain of your theme/plugin. Example:

```
/*
* Plugin Name: Post Reading Time
* Text Domain: post-reading-time
* Domain Path: /languages
*/
```

Here's the tricky part: it's not always needed. First of all, it's not required if you load your MO files with load_plugin_textdomain() because this function expects the text domain as its first parameter. The situation is a little different if you're using 'Domain Path'.

When auto-loading translations with 'Domain Path', the 'Text Domain' is required if your text domain doesn't match your theme's or plugin's slug. What is a slug? In practice, it's just the name of your main folder.

For our plugin located at /wp-content/plugins/post-reading-time/, the slug is "post-reading-time". That's why I used this awfully long text domain in all the i18n function calls - it's recommended to always use the slug as your text domain.

If your text domain was different from the slug (e.g., "pgn-reading-time"), then you'd have to specify it with "Text Domain: pgn-reading-time". Otherwise, the MO files wouldn't be loaded, even if you defined the Domain Path.

Inner Workings Of The Translation System

You've seen the "what" of translations. You know how to internationalize your code, you know how to translate it, and you know how to load the translations. It's time for you to understand the "how" behind this entire system.

The first step WordPress takes is determining what language to use. It usually does this by reading the WPLANG option (from the wp_options table). This option stores the active locale (e.g., en_US, pl_PL, fr_FR, etc.). This locale can be set in Settings.

Next, the appropriate MO file is loaded. The expected filename is computed, i.e., {text-domain}-{locale}.mo. If this file is found, it is parsed and loaded into memory. Remember that .mo files are just binary files containing the original string and its translated version.

The i18n functions, like __(), start by getting the appropriate Translations object. They get it from a global array of all loaded translations: $l10n. This is an associative array where the key is the text domain and the value is the object.

Every Translations instance contains an associative array of all translated strings where the original string is the key and the translated string is the value, like:

```
public $entries = array( $msgid => $msgstr );
```

That's a map of all the original hard-coded strings and all of their translations. Getting the translated value returned by __() is as simple as doing $entries[$msgid]. This class is defined in /wp-includes/pomo/translations.php if you want to have a look.

Code Example

Now that you understand how this system works, let's translate our Post Reading Time plugin to Polish as an example.

Internationalization

We'll start by internationalizing it, which means wrapping all strings in i18n functions. We'll walk through it step by step.

```
function pgn_add_snippet_to_post_content( $content ) {
    if ( ! is_single() || ! is_main_query() || ! in_the_loop() ) {
        return $content;
    }

    $reading_time_text = get_option( 'pgn_reading_time_text' );
    if ( empty( $reading_time_text ) ) {
        $reading_time_text = __( 'Reading time', 'post-reading-time' ); // Default
    }
    $reading_time = pgn_calculate_reading_time( $content );

    $mins_text = _n( 'min', 'mins', $reading_time, 'post-reading-time' );

    $display_text = sprintf(
        /* translators:
         * 1: The label, either default ("Reading time") or user-defined
         * 2: The number of minutes
         * 3: The word "min" or "mins"
        */
```

```
        __( '%1$s: %2$d %3$s', 'post-reading-time' ),
        $reading_time_text,
        $reading_time,
        $mins_text
    );

    $display_html = '<div class="pgn-reading-time">' . $display_text . '</div>';
    $display_html = apply_filters( 'pgn_display_html', $display_html,
$reading_time_text, $reading_time );

    return $display_html . $content;
}
```

There are a couple of interesting things going on here. The first is obvious - the default "Reading time" got wrapped in __(). We'll now be able to translate the default string. But if you look closer, you'll see that it's only used when there was no string stored in the database. How do we translate the user-provided text? We don't.

There is no way to translate anything in the database, including any options or content, using MO files. That is not their job. Their job is translating hard-coded strings. We'll talk about translating content later.

The next one after the default reading text is $mins_text. We use the _n() pluralization function to be able to use a different word depending on the number of minutes ("min" vs "mins"). In this case, we made the 'mins' a separate variable because of how the final text is created.

The display text uses sprintf() with 3 variables: the label, the number of minutes, and the pluralized 'mins'. As you can see, we also included a much needed comment for the translators, explaining what those variables are.

```
function pgn_text_section_html() {
    echo '<p>' . __( 'Settings related to text', 'post-reading-time' ) . '</p>';
}

function pgn_settings_init() {
    // First is $option_group, second is $option_name (the one stored in the
database)
    register_setting( 'pgn_post_reading_time_group', 'pgn_reading_time_text' );

    add_settings_section(
        'pgn_text_section',          // $id - unique ID of the section
        __( 'Text Settings', 'post-reading-time' ), // $title - displayed as heading
```

```
        'pgn_text_section_html',     // $callback
        'pgn_post_reading_time_menu' // $page - slug of our menu page
    );

    add_settings_field(
        'pgn_reading_time_text',                // $id
        __( 'Text alternative to "Reading time"', 'post-reading-time' ), // $title -
label
        'pgn_reading_time_text_html',           // $callback
        'pgn_post_reading_time_menu',           // $page
        'pgn_text_section'                      // $section
    );
}

function pgn_annoying_welcome_message() {
    $text = __( 'Thank you for downloading my plugin!', 'post-reading-time' );
    echo '<div class="notice notice-info is-dismissible"><p>' . $text .
'</p></div>';
}
```

These two are just simple __() calls so I bundled them together. Nothing fancy or new here. You can see that we're internationalizing only the user-facing strings.

```
function pgn_settings_page_init() {
    add_options_page(
        __( 'Post Reading Time Settings', 'post-reading-time' ), // $page_title -
displayed in <title>
        _x( 'Post Reading Time', 'settings submenu title', 'post-reading-time' ),
// $menu_title
        'manage_options', // $capability - don't worry about that now
        'pgn_post_reading_time_menu', // $menu_slug - slug & unique ID
        'pgn_settings_page_html'      // $callback
    );
}
```

Last but not least, we use _x() for the submenu name, giving the string context. I did that mostly for demonstration purposes, but you could argue that the string "Post Reading Time" might be translated differently for the frontend, the plugin's name, and the submenu. This ensures that we can have a separate translation for the name of the submenu leading to our settings page.

Generating The POT File

Let's now head to our plugin's directory and run "wp i18n make-pot .". This will create a "post-reading-time.pot" file. Next, create the /languages directory and move this POT file there. Here are the contents of this file (minus the metadata at the top):

```
#. Plugin Name of the plugin
#: post-reading-time.php
msgid "Post Reading Time"
msgstr ""

#: post-reading-time.php:38
msgid "Reading time"
msgstr ""

#: post-reading-time.php:42
msgid "min"
msgid_plural "mins"
msgstr[0] ""
msgstr[1] ""

#. translators: 1: The label, either default ("Reading time") or
user-defined 2: The number of minutes 3: The word "min" or "mins"
#: post-reading-time.php:50
#, php-format
msgid "%1$s: %2$d %3$s"
msgstr ""

#: post-reading-time.php:65
msgid "Settings related to text"
msgstr ""

#: post-reading-time.php:80
msgid "Text Settings"
msgstr ""

#: post-reading-time.php:87
msgid "Text alternative to \"Reading time\""
msgstr ""

#: post-reading-time.php:113
msgid "Post Reading Time Settings"
msgstr ""
```

```
#: post-reading-time.php:114
msgctxt "settings submenu title"
msgid "Post Reading Time"
msgstr ""

#: post-reading-time.php:123
msgid "Thank you for downloading my plugin!"
msgstr ""
```

You can see that most of those entries are just simple key-value pairs with a comment at the top showing their origin. The msgid is the original string, and we'll fill msgstr later when translating. The first deviation is this:

```
#: post-reading-time.php:42
msgid "min"
msgid_plural "mins"
msgstr[0] ""
msgstr[1] ""
```

That's the entry for our _n() call. msgstr[0] is the singular translated string, and msgstr[1] is the plural one. There's another interesting entry right after:

```
#. translators: 1: The label, either default ("Reading time") or
user-defined 2: The number of minutes 3: The word "min" or "mins"
#: post-reading-time.php:50
#, php-format
msgid "%1$s: %2$d %3$s"
msgstr ""
```

That's how translators will see our comments. Now try to translate this msgid without it. Good luck. You can also see the "php-format" hint, indicating that we're using the sprintf function.

```
#: post-reading-time.php:114
msgctxt "settings submenu title"
msgid "Post Reading Time"
msgstr ""
```

And here's the _x() call. The context is stored as msgctxt. We're still only translating the msgstr, but the context makes it separate from other msgid equal to "Post Reading Time". One such entry was the plugin name at the very top of the file.

As a matter of fact, if we replaced this _x() call with a __() call, this entry would cease to exist. It would be merged with the plugin name, as the msgid (the original string) is exactly the same. Think back to how translations are stored in memory. For _x(), the key searched for in the array is actually "{msgctx}\4{msgid}", so it'd be "settings submenu title\4Post Reading Time" (\4 is a special ASCII EOT character).

Localization

It's time to create the PO file with the final translations. Let's start by translating the boring part - the __() and _x() strings. They are just a simple string. After that, we'll take care of mins and the display text with variables. Remember to leave the .pot file as is. Make a copy of it named "post-reading-time-pl_PL.po".

```
#. Plugin Name of the plugin
#: post-reading-time.php
msgid "Post Reading Time"
msgstr "Czas Czytania Wpisu"

#: post-reading-time.php:38
msgid "Reading time"
msgstr "Czas czytania"

#: post-reading-time.php:65
msgid "Settings related to text"
msgstr "Ustawienia związane z tekstem"

#: post-reading-time.php:80
msgid "Text Settings"
msgstr "Ustawienia tekstu"

#: post-reading-time.php:87
msgid "Text alternative to \"Reading time\""
msgstr "Alternatywa dla \"Czas czytania\""

#: post-reading-time.php:113
msgid "Post Reading Time Settings"
msgstr "Ustawienia Czasu Czytania Wpisu"

#: post-reading-time.php:114
msgctxt "settings submenu title"
msgid "Post Reading Time"
msgstr "Czas Czytania Wpisu"
```

```
#: post-reading-time.php:123
msgid "Thank you for downloading my plugin!"
msgstr "Dziękuję za zainstalowania mojej wtyczki!"
```

Localizing 'mins'

The boring part's done, it's time to tackle the more interesting translations. Let's start with translating the pluralized 'mins' string. We could do something like that:

```
#: post-reading-time.php:42
msgid "min"
msgid_plural "mins"
msgstr[0] "minuta"
msgstr[1] "minuty"
```

That would give use "1 minuta", "2 minuty", etc. But there's a huge problem. The Polish language, along with many other languages, has two different forms for the plural phrase… There's "2, 3, 4 minuty", and there's "5, 6, 7, 8, 9, 10, 11 minut". There are also exceptions, like the fact that for numbers 12, 13, and 14, the correct form is "minut", while for other number ending with 2, 3, and 4 (like 22, 23, 24), the correct form is "minuty". Here is the logic written in pseudocode:

if (number == 1) return "minuta"
else if (number.last_digit IN (2, 3, 4) AND number.last_two_digits NOT IN (12, 13, 14)) return "minuty"
else return "minut"

So how do you achieve that? Thankfully, the creators of gettext thought of that and introduced a special 'Plural-Forms' header. This header allows you to specify the number of plural indexes and a formula. Here's the header for Polish PO files:

```
"Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n%10>=2 && n%10<=4 &&
(n%100<10 || n%100>=20) ? 1 : 2);\n"
```

This achieves exactly the same thing as our pseudocode. Don't worry, you don't have to come up with that yourself. Every language has specified rules and you can easily find the correct formula for each of them. Thanks to this header, all _n() entries can now have 3 indexes, and the right one will be chosen according to the formula. Here's the final translation:

```
#: post-reading-time.php:42
msgid "min"
```

```
msgid_plural "mins"
msgstr[0] "minuta"
msgstr[1] "minuty"
msgstr[2] "minut"
```

PS: Because the _n() function only accepts a singular and plural argument, there is no way for you to use a language with 3 plural forms as the original language.

Localizing sprintf() Display Text

This one is going to be a breeze compared to _n(). Let me remind you of what we're working with:

```
#. translators: 1: The label, either default ("Reading time") or
user-defined 2: The number of minutes 3: The word "min" or "mins"
#: post-reading-time.php:50
#, php-format
msgid "%1$s: %2$d %3$s"
msgstr ""
```

Here's the thing - we could really leave it as is. Because there is no static text in this string, there's nothing for us to translate. All of the variables - the default text and the 'mins', have already been translated. They will only get inserted into this string.

The one thing we can do is change the order of the variables. In this case, it doesn't make any practical sense. The reading order in Polish is exactly the same as it is in English, so in reality, we'd just leave it. That being said, let's mess it up a bit for demonstration purposes:

```
#. translators: 1: The label, either default ("Reading time") or
user-defined 2: The number of minutes 3: The word "min" or "mins"
#: post-reading-time.php:50
#, php-format
msgid "%1$s: %2$d %3$s"
msgstr "%2$d %3$s - %1$s"
```

Now, instead of the string being "Czas czytania: 1 minuta", it'll be "1 minuta - Czas czytania". Notice that we're completely free to rearrange the flow of the sentence and delete, add, or modify any other static text - in this case, replacing the colon (:) with a hyphen (-).

Compiling & Loading The MO File

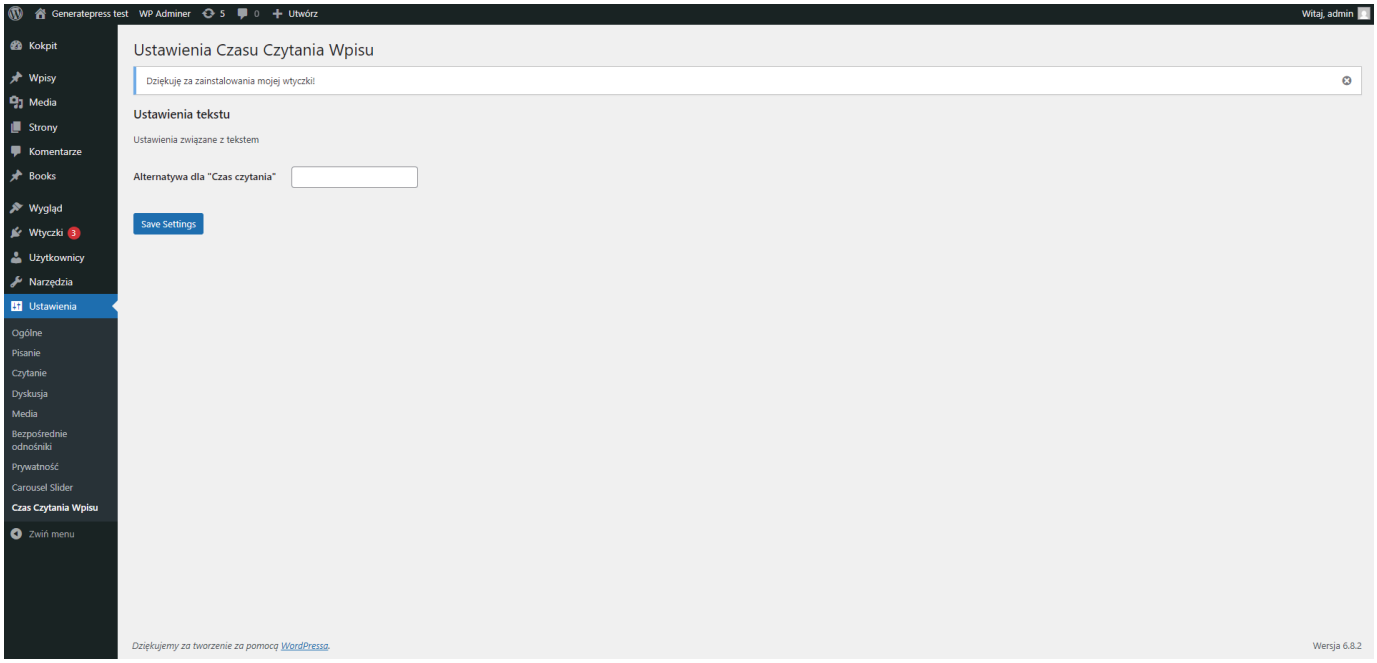
Now that we have the fully translated PO file, it's time to compile it and load it. You already know how to do that. Head to the /languages folder and execute:

```
msgfmt -o post-reading-time-pl_PL.mo post-reading-time-pl_PL.po
```

Then modify the main plugin file's header to include:

```
// Domain Path: /languages
```

That's it. Our translation file should now be loaded. You should see them if you change your language to Polish in Settings. Here's our translated settings page:



And here is the frontend of a post:

i18n In JavaScript

We've covered PHP, but what about strings in JavaScript? There are 2 ways to internationalize your js, and we'll start with the worse one.

wp_localize_script()

If you want to support WordPress <5.0, you have to use the wp_localize_script() function. This function accepts an associative array and creates a corresponding JavaScript object. You also need to pass it a handle (of your enqueued JS file) and the name for the object. The result is inline js rendered just before your script.

Note that you are in complete control over the passed array, which also means that you're solely responsible for making it contain all of the translations and using appropriate key names. You are not limited to any structure whatsoever. As a matter of fact, we've already mentioned this function before when discussing passing any arbitrary data from PHP to JS.

A standard way is to just hard-code all the keys in the array and make the values use the PHP translation system. This way, you can translate the strings in the PO files, even though they will eventually be used in JavaScript. Here's what I mean:

```
wp_register_script( 'my-plugin', plugins_url( 'my-plugin.js', __FILE__ ), [], '1.0',
true );

wp_localize_script( 'my-plugin', 'myPluginL10n', [
    'hello_world' => __( 'Hello world', 'my-plugin' ),
    'goodbye' => __( 'Goodbye!', 'my-plugin' ),
] );
```

The strings "Hello world" and "Goodbye!" will be translated on the server before the JS object is created. Then, in your JavaScript files, you'd have to remember to always use strings from this localization object:

```
console.log( myPluginL10n.hello_world );
```

JSON Files

WordPress 5.0 marked a new era of WordPress - the introduction of the Block Editor. Along with that came the beginning of a shift from PHP to JavaScript. The old and brittle translation system for JS was no longer a viable option, which is why core WordPress developers created the @wordpress/i18n package.

This package brings all of the most important i18n functions from PHP straight into JavaScript. This includes:

- __()
- _x()
- _n()
- _nx()
- sprintf()

Note the lack of _e(), as there's no echo in JS. Also note sprintf(), which is a native PHP function, but is so fundamental to the way the translation system works that it was included in this package. All of those functions have exactly the same signature and work exactly the same as their PHP counterparts.

Let's create an i18n-test.js file and load it in our plugin. Here's its code:

```
const { __, _x, _n, sprintf } = wp.i18n;
console.log( __( 'Hello, World!', 'post-reading-time' ) );
```

When loading a translated JS file, you have to specify a "wp-i18n" dependency. This is just a file handle of the JS file containing the package's code. You also have to call the wp_set_script_translations() function. This function expects, in order:

- The file handle of the enqueued script.
- The text domain.
- The full path to the directory containing your l10n files.

```
function pgn_load_js() {
    wp_enqueue_script(
        'pgn-i18n-test',
        plugins_url( 'i18n-test.js', __FILE__ ),
        array( 'wp-i18n' ),
        filemtime( __DIR__ . '/i18n-test.js' ) // auto-versioning
    );
    wp_set_script_translations( 'pgn-i18n-test', 'post-reading-time',
__DIR__ . '/languages' );
}
add_action( 'wp_enqueue_scripts', 'pgn_load_js' );
```

That's great, but how are these strings actually translated? Well, that's the interesting part. The WP-CLI command for generating the POT file parses JavaScript files as well. This means that all the __() calls in your JS files will be included in the POT file, among your typical PHP strings.

Because it's all in the exact same file, localization looks exactly the same. You just translate the string in the PO file. Here's the addition to our Polish translation:

```
#: i18n-test.js:2
msgid "Hello, World!"
msgstr "Witaj, Świecie!"
```

That's where similarities end. See, the JS translations do not use MO files. They use JSON files. More specifically, WordPress requires one Jed-formatted JSON file per JavaScript source file. Let me say that again. Every single one of your JS files will have its own translation JSON file. MO files are only for PHP.

These JSON files are the counterparts to MO files. You do not create or touch them directly. There's a new command specifically for generating these JSON files: "wp i18n make-json". You run this command after you've translated all your strings in the PO file. We'll run it like that:

```
wp i18n make-json languages languages
```

The first "languages" is the source directory, and the second "languages" (the same in this case, but that's not necessarily the rule) is the destination directory. The source directory is the directory containing your PO files. Here's the name of the file that'll be generated as a result of running this command:

post-reading-time-pl_PL-2879eac789a3f955c9bf014659d15893.json

And here is its content:

```
{
  "translation-revision-date": "YEAR-MO-DA HO:MI+ZONE",
  "generator": "WP-CLI/2.12.0",
  "source": "i18n-test.js",
  "domain": "messages",
  "locale_data": {
    "messages": {
      "": {
        "domain": "messages",
        "lang": "en",
        "plural-forms": "nplurals=3; plural=(n==1 ? 0 : n%10>=2 && n%10<=4
&& (n%100<10 || n%100>=20) ? 1 : 2);"
      },
      "Hello, World!": [
        "Witaj, Świecie!"
```

```
      ]
    }
  }
}
```

Okay, let's start with the obvious. What the hell is this filename? The answer is quite simple - it's the md5 hash of its script's filename. Think about it for a second. Every JSON file corresponds to precisely one JS file. They have to be linked together somehow, otherwise WordPress won't be able to find and load the appropriate JSON file. Using the filename itself is an option, but an even more robust approach is using its hash. Therefore, the JSON's filename format is: {text-domain}-{locale}-{md5_hash}.json

You shouldn't be very concerned with the JSON file itself. You're never going to have to read it directly. As I already said, it's the MO file equivalent for JS. You can see that the underlying logic is the same. There's a key-value pair object with original strings and corresponding translations. There's also our plural-forms header.

One interesting quirk of the make-json command is that, by default, it deletes all the JS translations from the PO files. That's right, when you run "wp i18n make-json", your translations disappear from the .po files. If you want to prevent that, you can use the --no-purge argument. In reality, you'd never really edit those files in the terminal to begin with, but we'll get to that in a second.

After you generate the JSON files and load them with wp_set_script_translations(), your strings should be localized. Here's the console on the Polish version of the site:

Internationalization Best Practices

There are a couple of important rules you have to keep in mind when internationalizing your strings.

Internationalize Entire Sentences & Paragraphs

In most languages, word order is different than in English. It is important to make it possible for translators to structure sentences naturally. This means internationalizing the entire sentence instead of only parts of it.

Think back to our "Reading time: 1 min" example. We wrapped the entire sentence in __() and sprintf(), which later allowed us to change the order of those variables. We wouldn't be able to do that if we just concatenated this sentence from individually translated strings.

Some languages might require a different order of sentences in a paragraph. A general rule of thumb is to split your i18n at paragraphs (one __() call per paragraph of text).

Assume Strings Can Double In Length When Translated

Some languages require twice as many words to express the same idea. If you can, try to test your layout for such a possibility.

Use The Same Phrases In The Same Forms

Try to benefit from the fact that the translator only has to translate an original phrase once for it to be used in multiple places. For example, try not to do that:

```
__( 'Posts', 'my-theme' );
__( 'Posts:', 'my-theme' );
```

If you do, the translator will have to translate "Posts" and "Posts:" separately.

Do Not Include URLs

If you include a URL in the original string, a malicious translator could make it a different URL. There's a good chance you wouldn't notice until it was too late. For that reason, you should always include URLs as placeholders (%s) using sprintf.

PHP Files Instead Of MO Files

WordPress 6.5 introduced significant performance optimizations to the server-side translation system. Namely, it became possible to store translations in PHP files instead of MO files.

Why is that a big deal? To get a long answer, you should go read the dev note for that. The short answer is that loading PHP files is much cheaper than loading and processing binary MO files. The translations are already stored in an array and you benefit from OPCache (if available).

Overall, it has been found that loading translations from PHP files improves the speed of your website by about 40 ms. That is *a lot* for such a small thing. Since 6.5, WordPress prioritizes those PHP files over MO files.

To generate PHP I10n files, you can run the "wp i18n make-php" command. This will generate the PHP files from your PO files. Their filename form is: {text-domain}-{locale}.l10n.php. If we

run this command, the generated file will be named post-reading-time-pl_PL.l10n.php. Here is its content:

```
<?php
return ['domain'=>'post-reading-time','plural-forms'=>'nplurals=3;
plural=(n==1 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 :
2);','language'=>'','project-id-version'=>'Post Reading
Time','pot-creation-date'=>'2025-08-28T09:59:52+00:00','po-revision-date'=>'
YEAR-MO-DA HO:MI+ZONE','x-generator'=>'WP-CLI 2.12.0','messages'=>['Post
Reading Time'=>'Czas Czytania Wpisu','Reading time'=>'Czas
czytania','min'=>'minuta' . "\0" . 'minuty' . "\0" . 'minut','%1$s: %2$d
%3$s'=>'%2$d %3$s - %1$s','Settings related to text'=>'Ustawienia związane z
tekstem','Text Settings'=>'Ustawienia tekstu','Text alternative to "Reading
time"'=>'Alternatywa dla "Czas czytania"','Post Reading Time
Settings'=>'Ustawienia Czasu Czytania Wpisu','settings submenu titlePost
Reading Time'=>'Czas Czytania Wpisu','Thank you for downloading my
plugin!'=>'Dziękuję za zainstalowania mojej wtyczki!']];
```

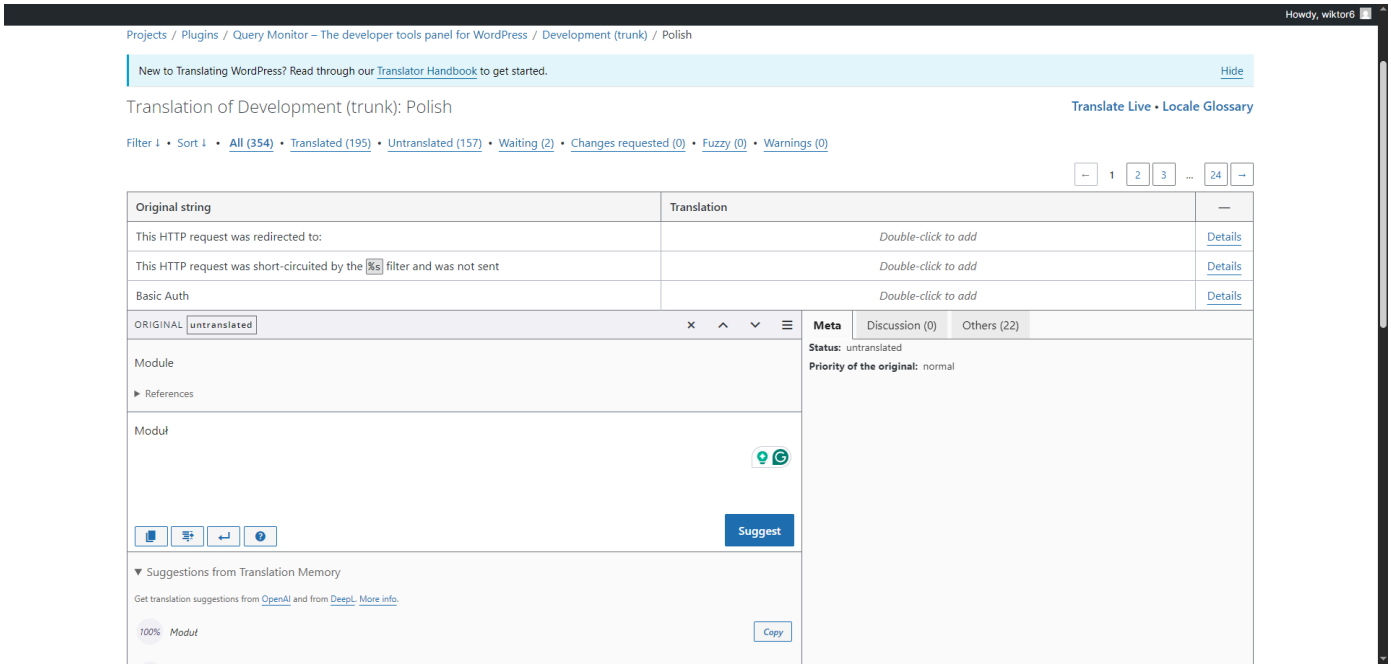
Alternatively, you can install the Performant Translations plugin. This is the project that started it all and it's still being developed. It automatically generates those PHP files for all themes/plugins that do not yet have them.

Hosting Translations On wordpress.org

Remember when I said that you wouldn't usually generate and manage translation files yourself? The reason for that is the fact that you will most likely want to host your themes or plugins in the wordpress.org theme or plugin repositories. If you do that, you can benefit from the translate.wordpress.org system.

This is an open, community-driven translation platform for all WordPress projects hosted on wordpress.org, including the WordPress core itself. It uses GlotPress under the hood, which is literally just a WordPress plugin. You can even download it on your own website and make your own collaborative translation platform.

The two most important user roles in this system are Contributor and Translation Editor. Anybody with a wordpress.org account can be a contributor. An editor is a trusted person responsible for approving translation suggestions made by contributors. Here is what this system looks like on the frontend for a contributor (translating the Query Monitor plugin):



What this means for you as a plugin/theme developer is that you theoretically don't have to pay a translator. Now, whether or not someone will actually be interested in translating your project is another question.

How It Works

One of the greatest blessings of this system is that you don't have to care about the l10n files at all. They are all automatically generated for you. The moment you upload your theme/plugin to the repository, the POT file is created by the system and contributors can translate it using the UI you just saw. Your only responsibility as a developer is internationalizing your code. You don't even have to include the 'Domain Path' header or call load_plugin_textdomain().

When you download a plugin in your admin dashboard, WordPress will automatically download the necessary translation files. It will only download files that are actually needed. This means that if your website is in English, no translations will be downloaded. But as soon as you change it to Polish, WordPress will automatically download all available translations for all your plugins and themes.

Another big change is where these translations are stored. They are **not** placed in the plugin's/theme's directory. They are placed in the /wp-content/languages directory. This folder contains all core translations, and it also has "plugins" and "themes" subdirectories.

Note that for a localization file to be automatically installed by WordPress from translate.wordpress.org, the translation has to be at least 90% complete. If your translations are only 50% done for a given locale, the files will not be downloaded, and your strings will be displayed in English.

Translating Content

Everything we've been discussing so far was related to translating the interface only. That is - the hard-coded strings. But that's not the only part of a website. Arguably, it's not even the largest one. The other side of this coin is content. This includes:

- post content and titles,
- post slugs,
- custom field values (metadata),
- taxonomies names and descriptions,
- options (like our alternative "Reading time"),
- and more…

I'm going to tell you something disturbing, so make sure you're sitting down. WordPress does not currently provide a native way of translating content. Content translation is achieved only with plugins. As a matter of fact, localizing content stored in the database is such an enormous undertaking that translation plugins are one of the most complex plugins in the entire Wordpress ecosystem.

There are 5 types of translation plugins, each working in a different way. We'll cover them one by one, along with their pros and cons.

1. Separate Post Per Language

This family of plugins is the most popular one, and for good reasons. The two biggest players in this category are WPML and Polylang.

It is arguably the most natural way of structuring translations. Every translation is a separate post with its own content. If you have an English and a Polish version of your site, you will have 2 different posts - one English and one Polish. Each post has its own row in the database, its own ID, its own URL, its own metadata, etc.

These plugins look at the URL and hook into different parts of WordPress. For example, if the path starts with /pl/, a plugin might filter every WP_Query call to make sure only translated posts are returned. It obviously also filters the global $locale variable to load the appropriate MO/PHP/JSON files for plugin and theme translations.

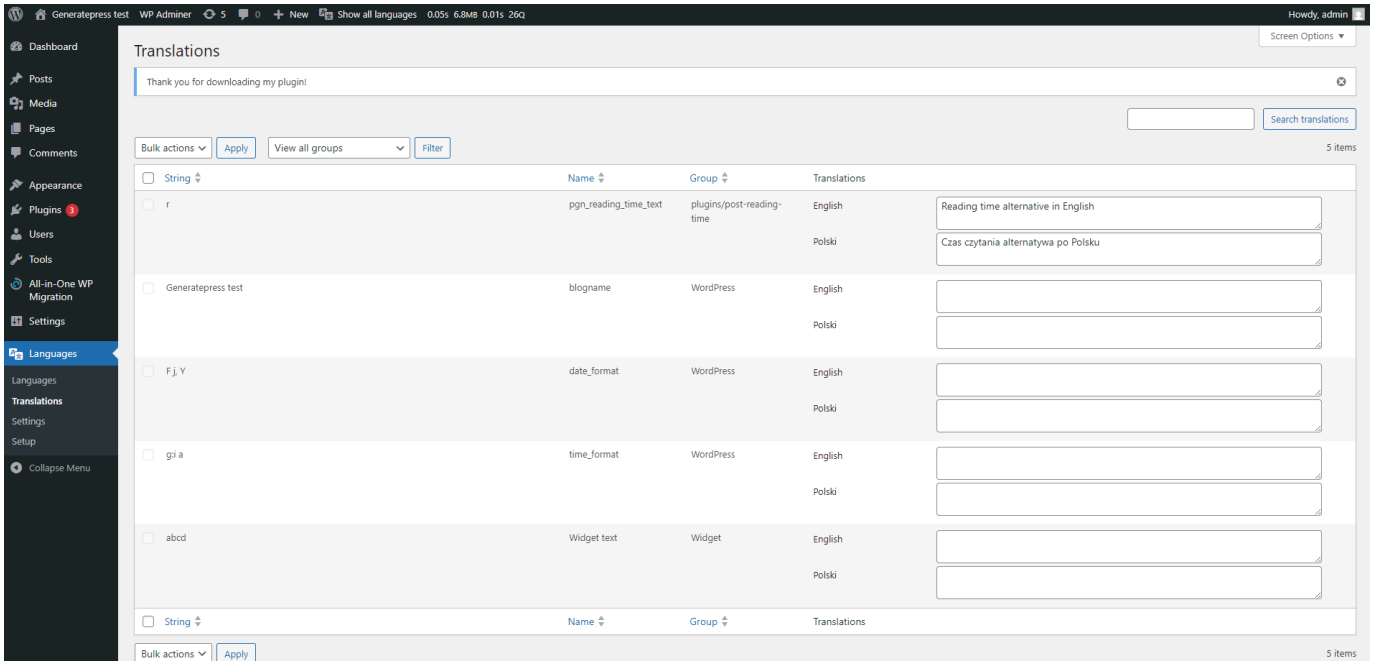
You can see the advice to use WordPress standard APIs shines again. If you were to, for example, use $wpdb directly instead of using WP_Query, these plugins wouldn't work as expected. Furthermore, plugins like that rely heavily on custom database tables. There's only so much you can reasonably achieve with WordPress's default schema. They usually also provide some kind of string translation.

Think about our alternative "Reading text" option in wp_options. That's not a post, so it's not as obvious how to make it translatable. To achieve that, usually a wpml-config.xml file is used. Yes, that's a format created by the WPML plugin, which has been around since 2007. It's become a

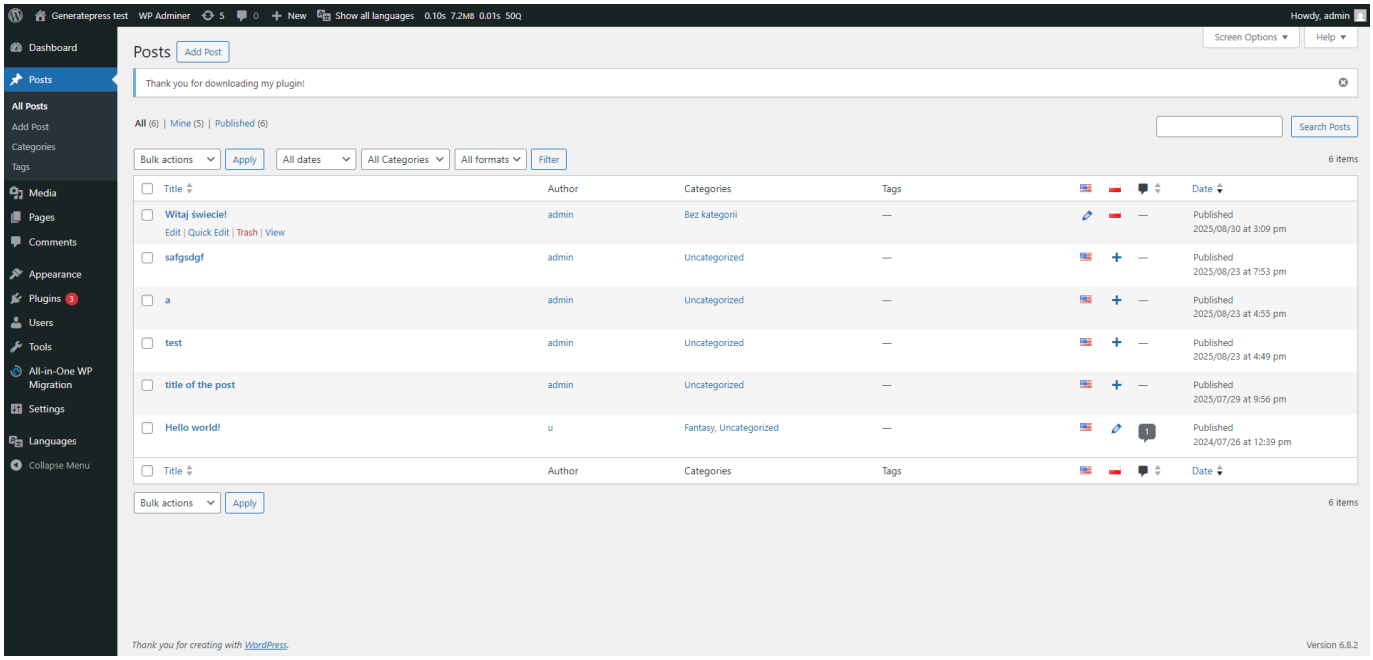
de facto standard. Here's the content of this file required to make our option translatable:

```
<wpml-config>
    <admin-texts>
        <key name="pgn_reading_time_text"></key>
    </admin-texts>
</wpml-config>
```

That informs these plugins that they should hook into every get_option() call with this key and filter its output with the translated version. It's the plugin/theme author's responsibility to provide this file in their root directory. It's used for many more things than just options, and you should read the [wpml-config.xml documentation] for more info. Here's what the translation interface for our option looks like in Polylang:



And here is what the "Posts" menu looks like with Polylang. Notice the buttons to add/edit versions for each active language:



Pros

The pros of this architecture are that it's mostly native to Wordpress's data model and usually provides good SEO control (unique URLs per language). It also has broad ecosystem support, as it depends on standard WP APIs. It's less invasive into the way WordPress works, which can prove very beneficial, especially considering compatibility with other plugins.

Cons

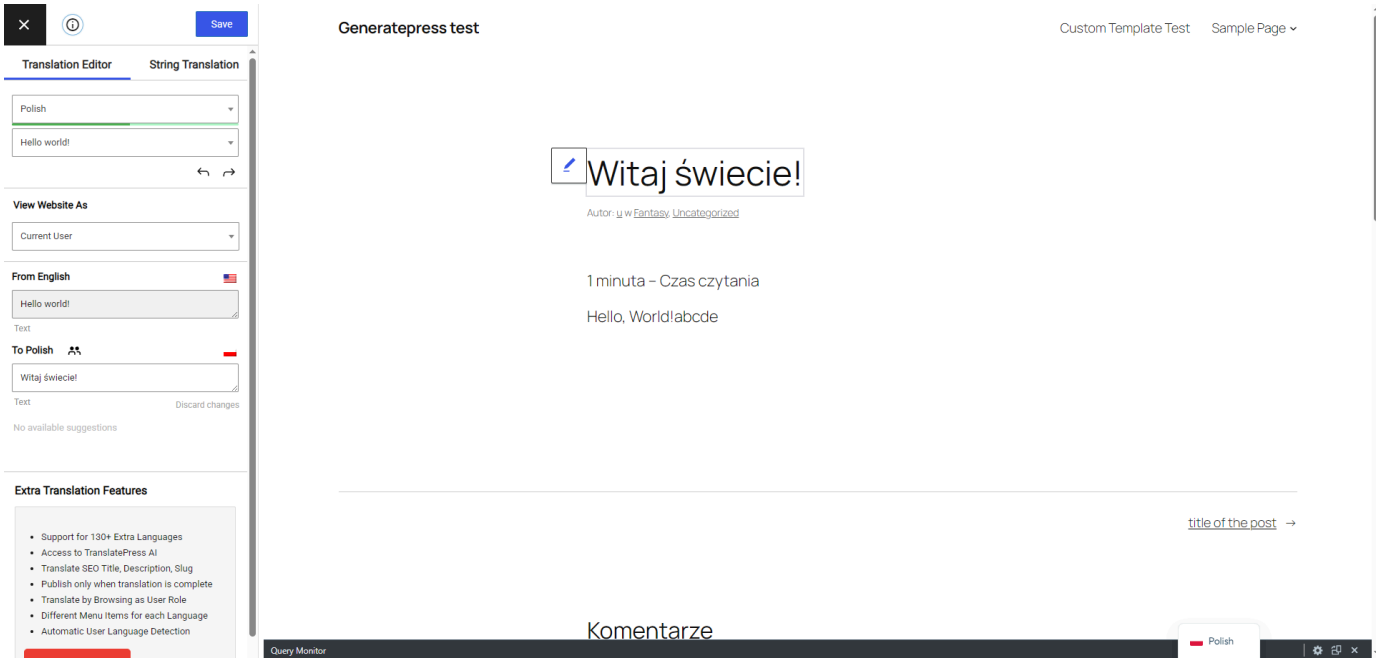
The biggest cons are related to the increased size of the database. You will have twice as many entries in many of your tables. The custom tables add additional complexity to the database schema.

In my opinion, this family of plugins is ideal for 95% of websites. It provides a perfect balance between simplicity of use and compatibility with WordPress. It's easy to understand and the effects of the increased database size are usually negligible.

2. String-Level Translation

Probably the next most common type of translation plugin is the "single page, string-level translation" plugin. The prime example is TranslatePress.

Plugins like that do not create separate posts for different language versions. They scan the content of the entire page and provide a proprietary frontend editor for translating what you see. It's a little hard to explain in words so take a look at TranslatePress's UI:



You can see that we are not translating the content in the editor, but while looking at it on the frontend. You can click on every string you see on the screen and translate it in the panel on the left. The translations are not stored as the entire content of a post. Instead, every singular identified string (a paragraph, a variable, etc.) gets its own row in the database.

All of the translations are stored in custom tables. The plugin hooks into the content while it's being displayed and swaps original strings for translations. There's no such thing as filtering the WP_Query to display only the translated posts, as there is only ever one post in the database. By default, everything is displayed, whether it is translated or not (you can manually exclude posts and pages from being displayed on certain language versions).

Pros

Plugins like WPML and Polylang are more intuitive for us as developers (at least they are for me). Their data model makes sense. Plugins like TranslatePress are more intuitive for normal people who do not understand how WordPress or databases work. They see the entire page and they can change it right there, on the frontend. It's natural, easy, and responsive. That is the biggest advantage of this type of solution - ease of use for non-technical people.

Cons

Unfortunately, the downsides are pretty stark, especially for larger or more advanced websites. Translations are fragile, as they are key-value pairs. If you change the original text, even by adding a comma, the key changes, and the translation gets invalidated. It's also harder to manage string-level translations for websites with hundreds/thousands of pages.

Another problem is compatibility. WPML and Polylang played nicely with the native WordPress way, but TranslatePress is more "hacky" - replacing strings in the rendered HTML. Let's say you have an SEO plugin. How will you make it work separately for the English and the Polish post? You can't because there's only one post in the database. This solution is overall more likely to be problematic and incompatible with other plugins.

It's also my assumption that TranslatePress is slower than "separate posts" plugins, as it requires additional computation to handle string replacement. I haven't conducted any tests on that though, so don't take my word for it.

What definitely is true is limited flexibility for different content per language. What I mean is not only simple translations, but possibly even separate layouts for different languages. Unlike with Polylang, you can't achieve that with TranslatePress. You are tied to string translations only. It's also harder to migrate TranslatePress translations to a different translation plugin.

That's 4 paragraphs for the downsides only, and you could definitely come up with a few more if you spent some time thinking about it. To be clear, I'm not totally against TranslatePress. Plugins like that work well for small to medium sites with no advanced needs, and when you need a live visual editor. Just keep in mind all of the above when making that decision.

3. SaaS Translation Proxy

There are solutions which are completely detached from WordPress. Some of the examples are Weglot and GTranslate. Instead of storing the translations in your website's database, they are stored on their servers.

The basic mode of operation is that either your website becomes a reverse proxy for the translation server, or the translation server becomes a reverse proxy for your website. Either way, every request to a translated page has to go through an additional network node - the server storing the translations.

Solutions like that tend to be CMS agnostic. They don't have to care if you're using WordPress or not because they don't interact with it. They only replace strings in the final rendered HTML. Kind of like TranslatePress, except even later in the lifecycle of the page.

They provide a frontend editing UI - very similar to the one you just saw with TranslatePress. They are usually more flexible, allowing you to modify even the raw HTML. The translations don't live in your admin dashboard, they live in the SaaS panel.

Pros

One of the biggest upsides of solutions like that is the fact that they tend to have significant support for automatic translations. TranslatePress supports automatic translations as well, but they are more embedded into the way these SaaS products work.

For example, GTranslate offers unlimited automatic translations without any additional charge. This can be a very important factor for certain kinds of websites. Imagine you have a website with a few thousand posts that get updated daily using an external API that doesn't provide translations. An automatic translation service is really your only option.

Cons

Having worked with GTranslate on such a website, let me tell you from experience - do **not** go this route unless you absolutely have to. Most of the TranslatePress downsides are also relevant to translation proxies.

On top of that, there are many more problems. The ongoing costs for SaaS products are usually much higher. There's limited control over what you can actually do in these panels. A lot of the time it doesn't work correctly. A number is substituted in the wrong place because a translated version uses a comma instead of a period, which makes your house price be $50 and the size be 500,000 m^2.

It's completely detached from your website. The biggest problem with that which nobody mentions is the fact that these solutions usually do not filter the global $locale in WordPress. This means that your plugins' and theme's l10n files will not be used! Your website will always be rendered in English and only translated by the SaaS. Your WordPress website has no idea there exists any translated version.

A huge problem stemming from that is JavaScript translations. If your JS renders any content on the page, this content is only rendered in the client's browser. It is not a part of the HTML returned by your server, which means that the translation proxy does not see it. Couple that with the fact that l10n JSON files aren't loaded, and you've got a recipe for disaster. It is a nightmare trying to translate JavaScript-added strings using services like that.

Oh, and did I mention that the translations aren't stored on your server, but instead on the service provider's server? Well, good luck pitching that idea to your lawyers, especially if you're in an industry handling sensitive personal data.

4. Separate Sites Per Language

There is one plugin that doesn't fit into any of the previous categories, and that's MultilingualPress. I'm not going to explain it in detail, because it's based on technology we haven't discussed yet (we will later). This technology is WordPress Multisite.

The basic idea is that each language version is a separate WordPress instance. A completely detached website in the same Multisite network. It provides unlimited freedom per locale (different themes, plugins, content, etc.). It's mostly fit for large sites and organizations, especially if a separate team handles each language version.

It's more work to manage and set it up. Cross-site syncing is more complicated. Generally speaking - for 99% of websites, it's like using a sledgehammer to crack a nut.

5. In-Content Markup (Legacy)

This last one is really more a fun fact than a useful translation architecture. It's a legacy system used by an old and abandoned qTranslate-X plugin (and its still supported community fork - qTranslate-XT). It provided string-level translations, but stored them in the post's content instead of in separate tables. The content might've looked like this:

[:en]Hello[:pl]Witaj[:]

You can imagine why this solution was so problematic. It uses the post_content column in a way it was never intended to be used. What if a plugin hooks somewhere to read or modify the content before qTranslate-X gets to parse it? It's a brittle solution. Don't use it.

Translating Block Themes

If you think back to blocks and block themes, you should remember that static blocks have hard-coded strings in their HTML content. This is a problem because HTML files can't be internationalized. It would be unacceptable for theme authors to ship themes that require plugins to translate their templates.

This is one of the reasons theme authors do not usually use static blocks in the templates or template parts they create. Instead, they make everything into a pattern, which is a PHP file. This PHP file can be internationalized using i18n functions like __(). You can go take a look at the official WordPress block themes, like the Twenty Twenty-Five theme. You will see patterns used everywhere.

User Accounts & Permissions

As you might imagine, WordPress takes care of all the stuff related to user accounts for you. This system is based on user roles and capabilities.

Roles

There are 6 default WordPress roles. They are ordered in a hierarchy, where each next role has all the capabilities of the previous role and more. Here they are:

- **Subscriber** - can only manage their profile.
- **Contributor** - can write and manage their own posts but not publish them.
- **Author** - can manage and publish their own posts.
- **Editor** - can manage and publish posts of other users.
- **Administrator** - has access to all administration features.

- **Super Administrator** - only relevant in WordPress Multisite. We'll cover it when discussing this topic.

You can add new roles using add_role(). You can remove roles using remove_role(). Your roles don't have to fit in this hierarchy. You can have a role which has some capabilities of an administrator, while simultaneously not being able to publish posts. Every user on your website has to have a role. A single user can have multiple roles.

Capabilities

Capabilities are the cornerstone of WordPress's user system. To understand how they work, you have to understand why they exist in the first place.

Let's say you want to display a banner at the top of your website only for users with certain roles. A naive approach would be to check the current user's role and compare them to some list of roles. This would work, but what happens when you add a new role and want to display that banner for it as well? You have to go into the code and modify the set of roles.

Capabilities solve this by decoupling the question "can this user do it?" from the question "what role does this user have?". In our banner case, instead of checking for a role, we would check the capabilities of the current user. Here is a code snippet:

```
if ( current_user_can( 'pgn_view_banner' ) ) {
    pgn_display_banner();
}
```

Every role has a set of capabilities assigned to it. If we now wanted to add a new role and make it see the banner, we would only have to add the "pgn_view_banner" capability while creating the role. The role is completely detached from the capability itself.

There are currently over 60 default capabilities in WordPress. Some of the most important ones are:

- edit_posts
- delete_posts
- publish_posts
- upload_files
- edit_other_posts
- install_plugins
- update_plugins

This is a short list just to give you a concept of what kinds of capabilities there are and how they are used by WordPress. The core is full of these capability checks. To see the full list of all

capabilities along with what default roles have them, see the official Roles and Capabilities Documentation.

Of course, there is no such default capability as "pgn_view_banner". We have to add it ourselves. It's just a string stored in the array of all capabilities associated with a given role. You'll see how that works under the hood soon.

PS: Capabilities can also be assigned directly to individual users using WP_User::add_cap().

Users In The Database

The user system utilizes 3 tables in the database: wp_users, wp_usermeta, and wp_options. Let's start with the first two.

wp_users contains all of the most important information about the user account. The most crucial columns are:

- **ID**
- **user_login** - plaintext login.
- **user_pass** - hashed password.
- **user_email** - plaintext email.
- **user_registered** - time and date the account was created on.
- **display_name** - public facing nickname.

There are obviously more things you want to store for your users, which is why **wp_usermeta** exists. It works exactly like wp_postmeta we covered when discussing Custom Fields. It even provides the same CRUD functions: add_user_meta(), update_user_meta(), get_user_meta(), and delete_user_meta(). It's used to store things like first and last name or description, but you can use it to store any arbitrary key-value pair associated with a user.

WordPress uses **wp_options** to store information about registered roles and their capabilities. Remember how registering a custom post type or a taxonomy was done only in code and not stored in the database? That's not the case for roles.

In practice, this means that functions like add_role() or WP_Role::add_cap() modify the database. They might have different behavior depending on the current state of roles/capabilities saved in the database, but the rule of thumb is to run them only once on hooks such as plugin activation. This ensures you're not making any unnecessary database calls on every page load.

The actual role info is stored in an option named wp_user_roles. The value of this option is a serialized associative array of roles and their capabilities. It looks something like this:

```
array(
```

```
    'administrator' => array( 'switch_themes', 'edit_themes', 'activate_plugins', ...),
    'editor' => array( 'edit_posts', 'publish_posts', 'upload_files', ...),
    // other roles....
);
```

We skipped over a crucial part - how roles are associated with users. This information is stored in **wp_usermeta** with the wp_capabilities key. Here is the value of this entry for a typical administrator user:

```
a:1:{s:13:"administrator";b:1;}
```

It's just a serialized array that would look like this: array('administrator' => true). But let's now do something interesting. Do you remember how I mentioned that you can assign capabilities directly to individual users? Let's create a new subscriber user and then run this code in functions.php:

```
function thm_add_user_cap() {
      $user = get_user_by( 'id', '3' ); // hard-coded ID for simplicity
      $user->add_cap( 'contributor' );
}
add_action( 'init', 'thm_add_user_cap' );
```

That's what the wp_capabilities value now looks like for this user:

```
a:2:{s:10:"subscriber";b:1;s:11:"contributor";b:1;}
```

What the hell? We added a capability and it looks and is stored exactly like a role! Let's write some debugging code to see how it works. We'll check if this user can edit posts, which is a capability assigned to the contributor role but not to the subscriber role. We will also check if the user has the capability "contributor" that we just gave them.

```
function thm_test_user_cap() {
      $can_edit = user_can( 3, 'edit_posts' );
      if ( $can_edit ) {
            echo 'can edit ';
      } else {
            echo 'cant edit ';
      }

      $can_contributor = user_can( 3, 'contributor' );
```

```
	if ( $can_contributor ) {
		echo 'can contributor';
	} else {
		echo 'cant contributor';
	}
}
add_action( 'init', 'thm_test_user_cap' );
```

Guess what gets echoed. "can edit can contributor". That's right. WordPress does not differentiate between a role and a capability. Both of them are just a string associated with the wp_capabilities meta key. Instead, if a capability happens to also be a role, all the other capabilities assigned to it (stored in wp_options) also get assigned to the user. A role is basically just an alias for a set of capabilities.

This means that you can check if "a user can administrator", which makes no semantic sense, and the return value will tell you if the user has the role administrator. That being said - treat this as a curiosity and not a useful tool. It's a non-standard way of using this function and is explicitly discouraged by the official documentation.

Sessions

WordPress does not use PHP sessions ($_SESSION) for logins. It uses signed cookies along with a server-side session token. This topic is a little advanced and requires some cryptography knowledge.

When you log into WordPress, it places two cookies on your browser - wordpress_logged_in_{COOKIEHASH} and wordpress_{COOKIEHASH} (or wordpress_sec_{COOKIEHASH}). Both of these cookies are a string with pipe-delimited pieces of data. The structure of the cookie is as follows:

username | expiration | token | HMAC

- **username** is the plaintext username of the account you're logged into.
- **expiration** is the UNIX timestamp of when the session is set to expire. The default session length is 48 hours or 14 days if the user clicks "Remember me".
- **token** is a random string.
- **HMAC** (Hash-based Message Authentication Code) is a cryptographically secure hash making this entire system secure.

An example cookie value might look like this:

```
admin|1756828649|b5mqL93RvtNKGfrSMOGnQibxNlhcVR4Ebtfvd5P0Oxk|fa881a1845f07b
```

```
6fb244f1f7f10e844ab70ce061306a1b28gb40b13fcaae92db
```

Let's untangle this step by step. First, what is COOKIEHASH in the name of the cookie? It's the md5 hash of your website's siteurl. If your siteurl, which is an option set in Settings and stored in wp_options, was https://example.com, the COOKIEHASH would be equal to "c984d06aafbecf6bc55569f964148ea3". This is used to ensure that sessions for different WordPress websites hosted under the same domain don't disrupt each other.

The token is basically just a random 43-character string generated for every session. The cookie you can see above contains its plaintext version. Its SHA-256 hashed version is also stored in the database in the wp_usermeta table.

The meta key is "session_tokens". The data is a serialized associative array of tokens (one per device), containing:

- the hashed token,
- expiration UNIX timestamp,
- user IP,
- user agent,
- login UNIX timestamp.

The HMAC is where the magic happens. Its computation is a little intense, but I'll do what I can to make it understandable. First, a string is created by concatenating the username, a fragment of the user's password hash, the expiration timestamp, and the token. The password fragment is just 4 characters of the password. Here's the string:

```
$username . '|' . $pass_frag . '|' . $expiration . '|' . $token
```

This string is passed to wp_hash() along with a scheme. A scheme is just a string with one of four values - "auth", "secure_auth", "logged_in", or "nonce". This is a good moment to answer a question that might've been bugging you for a while. Why does WordPress place two different session cookies and what's the deal with their names?

It's done to separate the /wp-admin cookie from the frontend cookie. The wordpress_logged_in_{COOKIEHASH} is placed on the root URL of your website. It is used to validate your session for all requests on the frontend. The wordpress_{COOKIEHASH} cookie (or the wordpress_sec_{COOKIEHASH} cookie if the website uses HTTPS) is placed on the /wp-admin path only.

The /wp-admin path has a different authentication cookie. What this means is that no frontend code (like JavaScript or requests to frontend URLs) ever sees the cookie that grants access to the admin panel. This is good. A vulnerability in any frontend code (e.g., a XSS attack) exposing the cookies would not allow the attacker to log into the /wp-admin panel, only to the frontend of

the website. By "frontend" I mean any URL that's not /wp-admin. By the way, every user gets an auth cookie, as even subscribers can update their profile in /wp-admin/profile.php

These cookies have the same contents, except for the HMAC. The reason they have a different HMAC is that they use different salts. Salts for each type of a cookie are defined as constants in the wp-config.php file. Example (shortened for readability):

```
define('AUTH_KEY', ' Xakm<o xQy rw4EMsLKM-?!T+,PFF})H4lzcW57AF0U');
define('SECURE_AUTH_KEY', 'LzJ}op]mr|6+![P}Ak:uNdJCJZd>(Hx.-Mh#Tz)p');
define('LOGGED_IN_KEY', '|i|Ux`9<p-h$aFf(qnT:sDO:D1P^wZ$$/Ra@miTJi');
define('NONCE_KEY', '%:R{[P|,s.KuMltH5}cI;/k<Gx~j!f0I)m_sIyu+&NJZ)-i');
define('AUTH_SALT', 'eZyT)-Naw]F8CwA*VaW#q*|.)g@o}||wf~@C-YSt}(d');
define('SECURE_AUTH_SALT', '!=oLUTXh,QW=H `}`L|9/^4-3 STz},T(w}W<I`.Jj');
define('LOGGED_IN_SALT', '+XSqHc;@Q*K_b|Z?NC[3H!!EONbh.n<+=uKR:>');
define('NONCE_SALT', 'h`GXHhD>SLWVfg1(1(N{;.V!MoE(SfbA_ksP@&`+A');
```

Every scheme (auth, secure_auth, logged_in, and nonce) has an associated {scheme}_KEY and {scheme}_SALT constant. The final salt is really generated by concatenating these two constants, so you can think about them as halves of the final salt. I'm not entirely sure why there need to be two constants but let's roll with it.

This is where we go back to our wp_hash() call with the string containing $username, $pass_frag, $expiration, and $token. I mentioned that the scheme is passed to this function. It is then used to get the correct salt. Again - this salt is going to be different for logged_in and for auth, which is what makes the frontend cookie different from the admin cookie.

The wp_hash() function computes and returns an md5 hash, with the data being the concatenated string, and the salt being the salt defined in wp-config.php. This hash becomes a key used to compute the final HMAC. Let me reiterate that. The hash generated with wp_hash() is **not** the HMAC. It is the key *used* to generate the HMAC.

The HMAC is computed very similarly, except it uses SHA-256 instead of md5. The hashed data is another concatenated string, this time without the password fragment:

```
$username . '|' . $expiration . '|' . $token
```

Again, the secret key used to generate this HMAC is the hash returned by wp_hash(). Notice that it is the only secret in this equation. If you knew the output of wp_hash(), you could compute the HMAC for any set of username|expiration|token parameters.

So what's making this hash secure? The 4 characters of the password add some marginal security, but their main purpose is to invalidate all sessions when the user changes their

password. What is really meant to be the secret are the salts. That's right - the constants defined in wp-config.php.

This might come as a surprise if you know anything about cryptography, as salts aren't usually meant to be secret. In reality, WordPress calls them "salts", but they are really secret keys. It's not the only confusing part of WordPress's terminology, which you'll soon find when we discuss nonces.

In practice, this has a few significant implications. First of all, your wp-config.php file is *very* important. If someone gains access to those secret keys, they can have a good shot at forging the HMAC of a session. They would then, for example, be able to compute the /wp-admin cookie using the data from the frontend cookie. Not to mention that this file stores your database credentials. Make sure to keep it secure.

The Authentication Lifecycle

Let's summarize the way sessions work in WordPress by looking at every step in order.

When you log in:

1. WordPress checks the username and compares the password hash with the hash stored in the database (in wp_users).
2. WordPress generates a random token and stores its SHA-256 hash in the wp_usermeta along with its expiration, IP, UA, and login timestamp.
3. Your username, expiration, token, and 4 characters of your password are used to compute a hash key.
4. The salts stored in wp-config.php are also used to compute that hash key. They provide security. They are different for the admin cookie and the frontend cookie.
5. The hash is used, along with your username, expiration, and token, to compute the final HMAC (again, different for admin and frontend).
6. The cookies are returned to the browser in a Set-Cookie header. The frontend cookie is placed on the root of the domain (/), and the admin cookie is placed only on /wp-admin.

When you visit a page logged in:

1. WordPress parses the cookie and checks if the expiration timestamp is in the past (the cookie has expired). If so, it gets invalidated.
2. WordPress independently computes the HMAC with the username, token, and expiration from the cookie, and the 4 character password fragment read from the database. The same salts from wp-config.php are used.
3. If the HMAC doesn't match, the authentication is unsuccessful.
4. WordPress checks if the token's SHA-256 hash is present in wp_usermeta. If not, authentication is unsuccessful.
5. WordPress sets the global $current_user object.

Security

We've actually already started discussing security when we covered its first and most fundamental layer - user permissions. In this chapter, we'll go deep into the most common vulnerabilities you can come across in WordPress and how to prevent them.

Common Web Vulnerabilities

Web security is broad. There are entire books devoted just to this topic. It would be impossible and counterproductive to cover it all. As such, I'll only brief you on some of the most common vulnerabilities you can encounter and that WordPress APIs are supposed to protect you against.

A quick sidenote before we begin: if you're writing a plugin or any other piece of code, please make sure to first understand what attack vectors your solution is susceptible to. Most vulnerabilities don't come from programmers forgetting to write protection - they come from programmers not knowing they need protection (i.e., incompetence).

A good first step in that is to just ask an AI chatbot. Let's say you were writing a plugin allowing users to upload files on the frontend. Explain this functionality to the chatbot and ask about attack vectors. It would quickly give you a list: malicious file upload, path traversal attacks, DoS via large uploads, etc. Whatever you do, just remember the most important rule: don't be stupid.

XSS

A cross-site scripting (XSS) attack is an attack consisting of injecting malicious JavaScript code on a website for it to get executed in the victim's browser. This can lead to the attacker gaining access to the victim's session cookies, theft of data, impersonation, redirection to malicious sites, and more.

Perhaps the most famous XSS in history is Samy, the MySpace worm. It was designed to display the string "but most of all, samy is my hero" and send a friend request to its creator (Samy Kamkar) from the victim's account. It would also further post the malicious payload on the victim's profile page, making it replicate itself. If you viewed such a page, the JavaScript would execute in your browser and you'd have become the next victim.

Vulnerabilities like that are caused by not sanitizing user-provided data. Let's stay in the realm of social media. Every social media has a way to post something, which is basically just a textarea. If the website didn't sanitize the text you put there, you could just write "<script>alert('hi')</script>", and every person that viewed your post would involuntarily execute this code.

CSRF

A cross-site request forgery (CSRF) attack consists of tricking the victim's browser into taking an unwanted action on a website they are logged into. This one is a little harder to explain or understand than XSS, but I'll try my best.

Let's say you were logged into a WordPress website example.com. You already know what it means - you have session cookies (e.g., wordpress_{COOKIEHASH}) stored in your browser. WordPress knows it's you by verifying this cookie.

Now, let's say someone knows you're an administrator for this website. They send you a phishing email that looks just like the newsletter you usually read. You click a link from this email, but instead of taking you to the newsletter's website, it takes you to a malicious website controlled by the attacker.

Remember, every website you open can run any arbitrary JavaScript code in your browser. This one is simple. It sends a POST request to example.com/wp-admin/options-general.php?page=pgn_sett. It just so happens that you're using a plugin with a vulnerable custom settings page located under that URL.

So what do you think happens next? Well, I'll tell you. The browser makes a request to the example.com/wp-admin URL and includes all the cookies it has for this path. This means that the wordpress_{COOKIEHASH} administrative cookie will be included in this request to the server, even though it originates from a malicious JavaScript code.

WordPress doesn't see any difference. It validates the cookie, and if it isn't expired, it sets the user and the permissions. You're an administrator, so you can change the options for the plugin. WordPress parses the POST payload (created and sent by the attacker) and saves the changes. The attacker was able to force you to take an action on a website you were logged into without you or them ever visiting this website directly.

A CSRF vulnerability can be prevented by including a unique token in the rendered HTML form and then checking if the POST request contains this token. The attacker doesn't have access to this token as it's generated by the server for every form that allows you to take some action (the settings page in our example, which is never accessed by the attacker before making the request).

PS: The example above was based on JavaScript, but a CSRF attack does not require JS. It can be achieved even with basic HTML, like by putting an action link in an image's 'src' attribute. The point is that browsers automatically include cookies, even with cross-site requests.

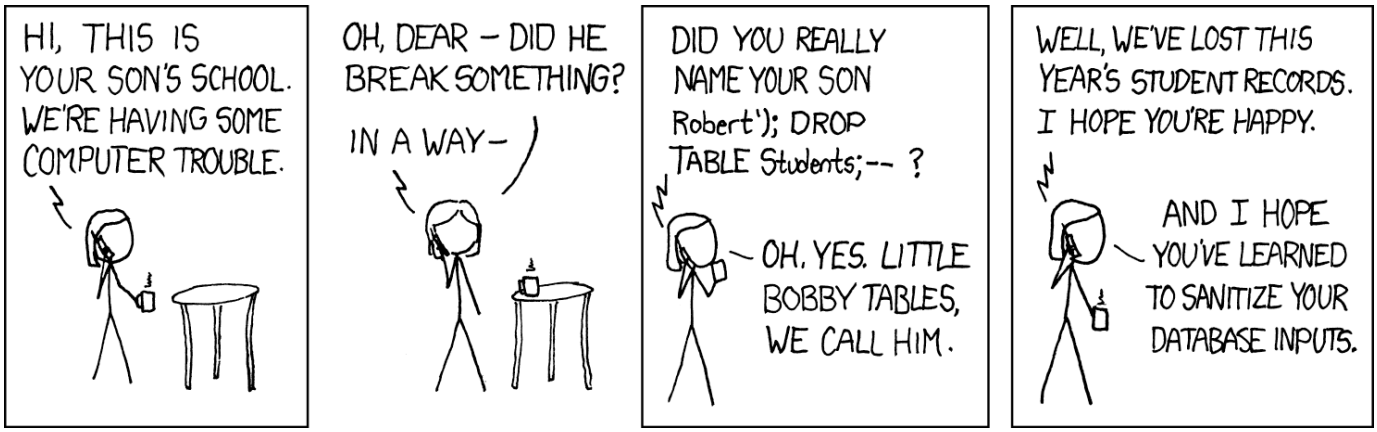
SQL Injection

SQL injection is similar to XSS, except it happens on the server instead of on the frontend. It consists of the attacker including a rogue SQL query in the request data.

Let's say you had a search engine on your website allowing users to look for a post by its title. A simplified SQL query would look something like this: "SELECT * FROM wp_posts WHERE post_title = $search;". If you ran this query, you'd be fried.

How about we search for a post named "abc;DROP TABLE wp_posts". Well, the final SQL query will be: "SELECT * FROM wp_posts WHERE post_title = abc;DROP TABLE wp_posts;". Do I really have to explain what's wrong here? Your wp_posts table will be dropped!

An SQL injection is perhaps the most notorious web vulnerability, especially among beginners. It's so tempting to just include the user-provided data in the query. It's also arguably one of the most dangerous vulnerabilities out there. You mitigate it by properly sanitizing the data when creating the final query.



Source: xkcd Exploits of a Mom

Security Mindset

Before we discuss securing your WordPress themes and plugins, it's important to start more broadly - with developing a security mindset.

The official WordPress Security Documentation proposes 3 rules:

- **Don't trust any data** (even the data already stored in your database).
- **Rely on the WordPress API** (most core functions handle security for you).
- **Keep your code up to date** (new security holes get discovered all the time).

These are excellent rules to keep in mind, but I think they forgot about the most important one - make security your top priority. Let's say you're creating a theme or a plugin. If you're successful, your code will run on hundreds, thousands, or even millions of websites world wide.

You are personally responsible for making sure these websites are secure. If you screw up, consequences can be astronomical. An attacker may steal personal data and scam hundreds if

not thousands of people - all because of your mistake. The owner can lose their business. People can lose everything they have.

My college professor once said that a single IT professional can sink an entire company. I think it's an understatement. They can sink lives. Think about it and do your fucking job.

Validation

Validation is the most secure way of protecting your website from rogue data. It's just a simple check resulting in one of two conclusions: valid or invalid. If the data is valid, you proceed. If it's invalid, you discard it.

It is secure because it's strict. Only the data you expect gets through. An example of validation can be checking if a string is a valid email address, if the value of a field is an integer greater than zero, or if it's present in a statically defined array of allowed values.

There are a couple of types of validation:

- **Safelist** - compare data to a finite list of expected values.
- **Blocklist** - reject data from a finite list of known and untrusted values. This is rarely a good idea, as you can't predict every bad data.
- **Format detection** - check the format of the data (i.e., is it a valid email, phone number, zip code, etc.).

Validation should be done as soon as possible. It's usually done with custom logic code, although there are a few useful built-in functions, like is_email(), term_exists(), username_exists(), etc. Many checks, especially when using format detection, will require some regex using preg_match().

Sanitization

Sanitization is the next best thing after validation. It's the practice of stripping away anything that may be dangerous in the context you're sanitizing for. Its most important goal in 90% of cases is to prevent XSS attacks. Sanitization usually happens right after you receive the data and before you save it in the database.

There are over 20 sanitization functions built into WordPress. The full list is in the official Sanitization Documentation. I could just give you that list, but you'll forget the first one before you get to the last one. Instead, it's more productive to think about sanitization functions as being in one of four categories.

1. General Purpose

This is your go-to, 90% of use cases, text sanitization category. It's for when you have some text and want it to be safe. There are only 2 functions here:

- **sanitize_text_field()**
- **sanitize_textarea_field()**

sanitize_text_field() is the most basic text sanitization function. It's pretty ruthless. You use it for single-line text content, i.e., a text input field. Here's what it does:

- checks for invalid UTF-8,
- converts single < characters to entities,
- strips all tags,
- removes line breaks, tabs, and extra whitespace,
- strips percent-encoded characters.

sanitize_textarea_field() does what sanitize_text_field() does except it preserves new lines (\n) and other whitespace. These are expected in textarea elements.

2. Specific Data Format

This category is for when the input has a strict, expected format. The goal isn't just to ensure safety, but to enforce a structure. The functions in this list are:

- **sanitize_email()**
- **sanitize_file_name()**
- **sanitize_hex_color()**
- **sanitize_html_class()** - creates a valid CSS class name from a string.
- **sanitize_key()** - leaves only lowercase alphanumeric characters, dashes, and underscores ("String Key" becomes "stringkey"). Useful for programmatic keys, e.g., meta keys.
- **sanitize_mime_type()**
- **sanitize_title()** - sanitizes a string into a slug ("My Amazing Post" becomes "my-amazing-post").
- **sanitize_user()** - sanitizes a username.
- **sanitize_url()**
- **absint()** - converts a value to a non-negative integer.

Most of these functions are self-explanatory. You use them only if you have specific requirements for the data. All of them do different things. Consult the documentation when using them.

3. HTML Content

This category is arguably the most tricky. It's for when you want to allow users to submit some HTML, like in a comment or a rich text editor. Once again, there are only 2 functions here:

- **wp_kses()**
- **wp_kses_post()**

What the hell does "kses" mean? I asked the same question the first time I saw it. According to a user comment from the wp_kses() documentation, this name comes from XSS, which is read similarly. It's also a recursive acronym for "**k**ses **s**trips **e**vil **s**cripts".

wp_kses() strips out all HTML tags and attributes except the explicitly allowed ones. Here's an example of calling this function:

```
$allowed_tags = array(
    'a' => array(
        'href' => array(),
    ),
    'strong' => array(),
);
wp_kses( $unsafe_content, $allowed_tags );
```

This will strip all tags except for "a" and "strong", and it will strip all attributes on those tags except for "href" on "a" tags.

wp_kses_post() is a shorthand for calling wp_kses() with all tags and attributes allowed in the editor. WordPress itself likely uses it to sanitize the content you input in the Block Editor. It's a good general sanitization function for when you want to allow rich text markup but want to ensure the content is secure.

4. Miscellaneous

This is the catch-all category. Those are rarely used functions that I didn't think were appropriate for the previous categories, either because they didn't fit their requirements or because they were so niche it'd be a waste of space putting them there.

Don't even bother trying to remember them unless you're megamind. Maybe it's good to read through them once in a while and think if you could use them in whatever you're writing.

The list:

- **sanitize_hex_color_no_hash()**
- **sanitize_meta()**
- **sanitize_option()**
- **sanitize_sql_orderby()**
- **sanitize_term()**
- **sanitize_term_field()**
- **sanitize_title_for_query()**
- **sanitize_title_with_dashes()**
- and more… (search for functions starting with "sanitize_")

Escaping

Sanitization happens right after input and before the data gets processed or saved in the database. Escaping happens right before the data is displayed. Why would you escape the data if you already sanitized it before storing it? Well, do you remember the first rule of security? Don't trust any data.

You should escape as late as possible - ideally right when the data is being outputted. There are a few reasons for that. The most important one is to prevent the data from changing after being escaped but before being rendered. This could introduce vulnerabilities. It's also much easier to maintain code when you don't have to wonder if that piece of data has already been escaped or not.

Similarly to sanitization, escaping functions are easiest to remember if you categorize them. In that case, the most sensible approach to categorization is by the place you're displaying the data in.

1. Plain Text

This is the most common use case. You have a piece of data and you want to display it "as is" (e.g., in a paragraph, in a header, in a div, etc.). The two functions you're looking for are:

- **esc_html()**
- **esc_textarea()**

You can see these are counterparts to the general text sanitization functions.

esc_html() converts all reserved HTML characters into their entities. It uses the PHP htmlspecialchars() function underneath with the ENT_QUOTES flag. What this means is that this function will replace these characters with these character sequences:

- **& => &**
- **< => <**
- **> => >**
- **" => "**
- **' => '**

What this does is it ensures the text will not be parsed as part of the HTML but as plain text. It means that if you do esc_html('bold text'), you will not see "**bold text**" on the frontend, you will see "bold text". If you didn't escape the string, echoing "<script>alert('xss')</script>" would result in JS execution.

If you sanitized a piece of data with wp_kses() and wanted to later display this data as a code snippet on the website, you'd use this function to actually show the tags in text. esc_html() is the most used escaping function. By the way, you can see how escaping differs from sanitization. The sanitization function removes the tags. The escaping function only encodes them.

esc_html() also passes $double_encode = false to htmlspecialchars(). It means that "&" does not become "&amp".

esc_textarea() is a confusing function. Semantically, you should always use it when escaping text in textarea elements. The only way it differs from esc_html() is that it doesn't check for invalid UTF-8 and it passes $double_encode = true to htmlspecialchars().

Double encoding is the primary characteristic. In most cases, it's the only pragmatic difference between this function and esc_html(). Consider what happens when a user enters the literal string "&" in a textarea. If you used esc_html() when rendering it, the user would see "&". That's a different string than the one they entered!

Double encoding ensures that this string gets returned as "&". The browser will convert the first entity to "&" and display the final string as "&" - exactly what the user entered. This is good, but notice that this is not a textarea-specific problem.

Let's say you had an input field on your website that allowed the user to store some string which would then be displayed on the page. If you display said string using esc_html(), and the user inputted a literal entity, the string would again be different than what they typed in. In this case, you'd have to use esc_textarea(), even though you're not escaping it inside a textarea.

This is the confusing part, and it's a misleading name of the function. In reality, esc_textarea() would better be called esc_html_double_encode() or something. The name is what it is because 99% of cases where you want to preserve the exact inputted text is with textareas. You want the person to see the content of what they wrote to be exactly the same after they refresh the page.

It's a little confusing so let me hone this point in using an example. Let's say you're making twitter. Every user has a textarea they use to write a tweet. A user types in "I love cats". When they save this tweet as a draft and refresh the page, they expect to be able to edit the exact same text they put in. That's what esc_textarea() does.

When they publish the tweet, two different things can happen. If you use esc_html() when displaying the post on their feed, the rendered content will be "I love cats". If you use esc_textarea(), the rendered content will be "I love cats". The use of this function is therefore not limited to textareas. You can use it whenever you want to render the exact text the way it's stored, including not converting HTML entities to their associated characters.

PS: Take note of what sanitization functions you use when storing the data in the database. Different functions might or might not convert characters to entities.

PS 2: Note that, technically, the content in the DOM is different from the content the user first inputted in the textarea - it contains the additional double-encoded entity. This is necessary to make it look the same. You have to coordinate your sanitization and escaping carefully,

otherwise each subsequent POST of the textarea form could add yet another layer of encoding to the stored content.

2. HTML Attributes

There is only one function for escaping HTML attributes - **esc_attr()**. You should use this function whenever you're outputting the value of an HTML attribute, such as 'alt', 'value', 'title', etc.

This function ensures that the data can't "break out" of the attribute. It achieves this by encoding the special HTML chars into their entities - exactly like esc_html(). As a matter of fact, the source code of esc_attr() and esc_html() is *identical*.

If these two functions are the same, why should you use esc_attr() instead of esc_html()? Because it's good practice. Their semantic meaning is different. Although unlikely, their implementations might change at some point, as they are used for achieving different things. It's just a coincidence that the current way of achieving them is the same.

Knowing that esc_attr() works just like esc_html(), you may sometimes find a need to use esc_textarea() instead (if you want to double encode entities). Treat esc_attr() as the baseline for escaping almost all HTML attributes, but use your best judgement when choosing the final function.

3. URLs

There is really only one URL escaping function - **esc_url()**. It ensures that the URL contains an allowed scheme (http, https, ftp, etc.). This prevents URLs like javascript:alert('XSS') from being output. If you just used esc_attr(), this wouldn't have been caught.

esc_url() also encodes special HTML chars to entities (&, ", ', etc.), just like esc_html() does. That's in accordance with the HTML specification. You should use esc_url() when rendering any URL anywhere. It doesn't matter if you want to display it as plain text or use it in a src or href attribute. Use esc_url(). It's an exception to using esc_attr() for any HTML attributes.

There's one more url escaping function you might come across - esc_url_raw(). The reason I didn't mention it earlier is because it's just an alias to sanitize_url(). It does what esc_url() does (validates the URL protocol), except it doesn't encode special chars to entities. It's not used for outputting, only for storing the URL in the database (which is the job of sanitization, so you should never really have to use this function).

4. Allowing HTML

When you want to allow some HTML to be processed as HTML, you should use one of these two:

- **wp_kses()**

- **wp_kses_post()**

Yes, these are exactly the same functions as for sanitization of content with HTML. They are universal sanitization and escaping functions. I'm not going to explain them again - read the sanitization section if you forgot them already (you goldfish).

Some people are going to tell you that you don't need to use wp_kses() again if you've already sanitized the content when storing it in the database. I don't agree with that. The reasoning, as far as I understand, is that it's not going to change the result in any way, and that wp_kses() is a rather computationally expensive function.

My argument for doing double wp_kses() is that there's never enough security. Think back to the first rule in the security mindset. Don't trust any data - even the one in your database. The skeptics will say "if someone hacked your database, you have a bigger problem than not escaping the content".

That is true, but what if no one has? Are you absolutely, unequivocally sure that you and everybody else on your team will remember to sanitize the data 100% of the time? How sure are you that the data in your database is actually sanitized? What if the code changes? A new junior maintainer on your team decides to rewrite the code handling the input and forgets to call wp_kses()? What then?

The point is - why open yourself up for a potential vulnerability, when you can just call wp_kses() again? Even if it takes additional 2 ms to render a page (which you will probably cache anyway), it's just not worth sacrificing the security.

5. Miscellaneous

Again, these are niche functions you will rarely use or see. It's good to know about them though:

- **esc_js()** - escaping inline JavaScript (like in the onclick attribute).
- **esc_xml()**
- **esc_sql()** - don't use it unless you have a good reason to. Read the section on database security below.

Escaping With Localization

There are some helper functions which allow you to localize the string before escaping it. Here they are:

- **esc_html__()**
- **esc_html_e()**
- **esc_html_x()**
- **esc_attr__()**
- **esc_attr_e()**
- **esc_attr_x()**

It's only a subset of the most commonly used escaping and i18n functions. They are simple wrappers. For example, here's the code of esc_html__():

```
function esc_html__( $text, $domain = 'default' ) {
	return esc_html( translate( $text, $domain ) );
}
```

Note that you can't use those functions on dynamic strings, as the gettext i18n system parses the code statically in order to generate the POT file (we've already discussed all of this in the i18n & l10n chapter). This means it's only useful for hard-coded strings.

The idea here is not to protect yourself from XSS attacks, as those would be impossible in a hard-coded string, but to encode the special chars to entities so that they don't break your HTML. That's why you should still escape the strings, even if they aren't read from the database.

Nonces

Nonces are used to prevent CSRF attacks. Let's start with a more general explanation of a nonce, and we'll get to the WordPress implementation later.

The name "nonce" comes from "**n**umber used **once**". It's supposed to be a random (or pseudo-random) one-time token used to verify a request. In its general form, nonces aren't tied to CSRF attacks only. They are a cryptographic tool used in many ways, such as preventing replay attacks or verifying data freshness.

Let's say you're rendering the HTML for an admin settings page (not in WordPress). You should create and include a nonce in a hidden input field for every form on that website. This way, when the user submits the form, you can check this nonce and compare it with the one stored in the database (you have to store the nonce after generating it).

If they match, it means that the form submission actually originated from the admin page and not from an attacker's website (i.e., a CSRF attack).

WordPress's Nonce Implementation

The explanation above was a quick summary of how "pure" cryptographic nonces work. WordPress's nonces are far from it. As a matter of fact, WordPress's nonce implementation is so unlike pure nonces that it's a common point of confusion. That's not to say it's inherently worse - it is not.

WordPress nonces are not used once. They are valid for a specified amount of time. They are also not numbers - they are md5 hashes. They don't get stored in the database - they are recomputed on every request.

That's cool, but you probably have no idea what any of that means. To truly understand it, we have to start with the "Why". WordPress was designed to be as stateless as possible. You should remember that from the chapter on user accounts. There is no session - the user is verified via the client-stored cookie.

A pure nonce implementation requires a lot of session-related architecture. Most importantly, nonces have to be stored in the database in order to be verified. If it's truly a random number, you can't reliably recompute it on the POST request. You have to save it in the database when it's created and then read it from the database to compare it with the nonce passed in the request.

This is a source of many problems. Let's say you had an admin page with 50 "delete" links. Every single refresh of that page would create 50 new rows in the database. That's a lot of additional operations and bloat in the database. You'd also probably need a new table specifically for nonces.

Every nonce would need some sort of expiration, but what happens if a user opens the same page in two tabs? Does the nonce from the older tab get invalidated or not? What if the user submits the form in tab 2 and then tries to submit it again in tab 1? What about back buttons? What about caching pages and serving them as static HTML? All of those are very painful problems when using pure nonces.

WordPress solved these problems by opting for a stateless implementation. Here's what happens, step by step, when you create a new nonce using wp_create_nonce($action):

1. WordPress gets the ID of the current user.
2. WordPress gets the current session token (passed in the user cookie).
3. WordPress gets the current tick for the action (we'll cover that soon).
4. WordPress creates a concatenated string: $tick . '|' . $action . '|' . $uid . '|' . $token
5. WordPress passes this string to wp_hash() with the "nonce" scheme and the default md5 algorithm.
6. WordPress returns a substring (10 characters) of the calculated hash.

The $action is just a hard-coded string. It's supposed to be different for every type of action. You can think of it as the nonce's name. The action of a nonce for a form saving some plugin settings might be "pgn_settings". We'll talk more about it later.

The "tick" is probably the most interesting part of this equation. It's also the hardest to explain. A WordPress nonce is valid for some amount of time. By default, the lifetime of a nonce is up to 24 hours. The current tick for the nonce is calculated using the wp_nonce_tick() function. Here's its source code:

```
$nonce_life = apply_filters( 'nonce_life', DAY_IN_SECONDS, $action );
return ceil( time() / ( $nonce_life / 2 ) );
```

time() returns the current Unix timestamp, e.g., "1757597729". It is then divided by half of the nonce's lifetime in seconds. Finally, it's rounded up. Let's visualize what this does on smaller numbers. Let's assume the current Unix timestamp is 21 and our $nonce_life is 10 seconds.

ceil(21 / (10 / 2)) = ceil(21 / 5) = 4. The current tick is 4. Another second passes, ceil(22 / 5) = 4. The tick is still 4. It's going to be 4 for the next 2 seconds, when time() is 23 and 24. The moment the Unix timestamp becomes 25, the tick becomes equal to 5. What this means is the nonce, which uses this tick for hashing, is going to become different (the concatenated string passed to wp_hash() will be different).

Now comes the most important part - verification. When you verify the validity of a nonce using wp_verify_nonce(), this function computes the hash for both the current tick (5) and the previous tick (4). That is why we're dividing the current Unix timestamp by only a half of the nonce's lifetime.

Why? Imagine what would happen if we requested a page with a form precisely when the Unix timestamp was 28. We'd get a nonce generated for the tick "4". We then submit this form a few seconds later, when the timestamp is 32. If wp_verify_nonce() only checked the current tick, our submission would be rejected! Imagine the mayhem if every form you loaded on the edge of the tick became invalid the second the next tick started.

Also, notice that our nonce was not valid for 10 seconds. It was valid for 9 seconds, because it first got generated at timestamp 21, not 20. That's why I said that, by default, a nonce is valid for *up to* 24 hours. In reality, it's between 12 and 24 hours - the length of 2 ticks (where the length of the first one can be shorter than 12 hours).

Don't feel stupid if you still don't understand it. It really is an advanced concept. Let's summarize it:

- When WordPress generates a nonce, it adds a tick to the hashed string.
- A tick is just an integer. It increases by 1 every time the rounded up result of (unix timestamp / half of the lifetime) increases. This means that if the lifetime of a nonce is 10 seconds, the tick will increase by one every 5 seconds (when the timestamp is 20, 25, 30, 35, etc.).
- This increase of the tick produces a different md5 hash every half of the lifetime of the nonce. That means the nonce becomes different.
- When verifying the nonce, WordPress computes its version both for the current tick and the previous tick.
- If WordPress only checked one nonce, generated with a tick of the full lifetime (10 seconds), all the nonces would become invalid the very second the result of (Unix timestamp / tick in seconds) changed. This would mean that if you refreshed the page when the timestamp was 29 and submitted the form 2 seconds later - when it was 31, the nonce would be invalid (even though only 2 seconds have passed). Instead, using the half-life of the nonce, its real lifetime is at least a half of the configured value.

This tick system eliminates the need for storing the expiration time of the nonce and storing the nonce in the database. Instead, the nonce is dynamically computed on every request, and its value automatically changes every half of the configured nonce's lifetime. Brilliant.

As for wp_hash() - we've already covered it when discussing user accounts and sessions. The "nonce" scheme means it uses the NONCE_KEY and NONCE_SALT keys from wp-config.php when computing the hash. These are the secret keys used to make the nonces secure. Without them, anybody would be able to compute the correct nonce knowing a user's ID, their session token, and the action (name) of the nonce.

To summarize: WordPress nonces are not real nonces, because they aren't used only once. As such, they don't protect you from replay attacks or other attacks the way pure nonces might. Their only job is to protect you from CSRF attacks. This is the only reason they exist. You should not use them for authorization or authentication. Always assume nonces can be compromised.

Creating Nonces

You now know the theory behind nonces. It's time you learn how to use them in practice. A nonce needs to somehow be included in the action you want to protect from CSRF attacks.

There are 3 functions used to create and/or output nonces:

- **wp_create_nonce()**
- **wp_nonce_field()**
- **wp_nonce_url()**

You already know **wp_create_nonce($action)**. It's the baseline function used for creating a nonce. It is used by the other 2 functions. It doesn't echo anything, it returns the nonce, like "bdf297b994".

The $action parameter, as we already discussed, is the unique name of the action the nonce is supposed to protect. You should strive to make it unique and specific. If you used the same action "delete_post" for all of your "delete" buttons, the nonce for deleting all posts would be the same. Compromising this one nonce would allow an attacker to delete all of your posts. It's better to make the action a dynamic string "delete_post_$id". This way, the nonce will be different for all your delete buttons.

wp_nonce_field() does the heavy lifting for you when you want to output the nonce in a form element. It accepts the action and the name of the input field, and echoes the field's HTML markup. By default, it also includes a field containing the referer, which is just the current request's URL (we'll talk more about it when covering verification). Calling:

```
wp_nonce_field( 'pgn_action', 'pgn_nonce_name' );
```

on the /wp-admin/options-general.php page results in this HTML markup rendered (the nonce is obviously going to be different):

```
<input type="hidden" id="pgn_nonce_name" name="pgn_nonce_name" value="d1195fa9c3">
<input type="hidden" name="_wp_http_referer" value="/wp-admin/options-general.php">
```

wp_nonce_url() adds a nonce as a query arg in the passed URL. It's for actions taken on GET requests (clicking a link) instead of form submissions, such as the aforementioned "delete" button (which would probably just be a link). Calling:

```
$url_with_nonce = wp_nonce_url( 'https://example.com/?query=1', 'pgn_action' );
```

will result in $url_with_nonce being equal to: https://example.com/?query=1&_wpnonce=d1195fa9c3.

The function adds the query string to the URL using add_query_arg() (a pretty useful wp function). If you're attentive, you might be wondering about something. Why is the ampersand (&) encoded? Well, that's because wp_nonce_url() calls esc_html() on the URL just before returning it.

That's right - a function responsible just for adding a nonce to a URL, escapes said URL before returning it. Not only is the concept itself stupid, but it doesn't even use the correct escaping function, which is esc_url(). Ticket #20771 on Make Wordpress Core from 2012 sheds some light onto this blunder. It seems to have been kept for backward compatibility. The bottom line is - use esc_url() on the URL returned by this function, even if the documentation states that it "returns an escaped URL".

Verifying Nonces

A nonce is useless if you don't check it before performing the action (in the code which actually does the thing, e.g., deletes the post). There are 3 functions designed for verifying a nonce:

- **wp_verify_nonce()**
- **check_admin_referer()**
- **check_ajax_referer()**

wp_verify_nonce() is the baseline function which recomputes the nonce and compares it with the one passed as an argument. It's used internally by the other functions. You can use it directly if you want to decide what happens when the nonce is incorrect.

check_admin_referer() is perhaps the most inaccurately named function in the entire WordPress core, and you'll see why I say that very soon. At its core, all this function does is it checks the nonce and calls die() if it's invalid.

It looks for the nonce in $_REQUEST[$query_arg], where $query_arg defaults to "_wpnonce" (the default name for wp_nonce_field() and default query args key for wp_nonce_url()). If you used a different name, like our "pgn_nonce_name" for wp_nonce_field(), you have to pass it as an argument. It works because the $_REQUEST superglobal contains values from both GET and POST requests (URLs and form submissions).

Here's the thing - you can use this function in code running on the frontend. It is *not* limited to the admin panel. As for the "referer" part - it's not even used in most cases. To understand where the name comes from, we have to dive into the genesis of this function.

check_admin_referer() was added in WordPress 1.2. Nonces, along with all the nonce-related functions, were added in WordPress 2.0.3. This function is older than nonces. Before nonces, what it did was it checked the referer header and called die() if the referer was not a URL inside /wp-admin. That is where the name comes from.

Right now, it only checks the referer in a very specific circumstance - when the nonce verification failed and the $action argument was not passed. If that's the case, and the referer isn't in /wp-admin, the function will call die(), just like its old version did. It's a backward compatibility quirk. By the way, the referer value now prioritizes the _wp_http_referer field you saw earlier (rendered by default when using wp_nonce_field).

Why they didn't create a new function with a more fitting name after introducing nonces is beyond me. It is one of the most confusing functions in WordPress precisely because of its bad naming. In its current form and in most cases - it has nothing to do with admin, nor with a referer. It just verifies the nonce and dies if it's invalid.

The dying part is true, except for one very specific edge case introduced by the backward compatibility check. See, if the nonce is invalid and $action was not passed, but the referer is indeed /wp-admin, then the function will not call die(). Instead, it will return the result of the nonce check, which is false.

This might be very confusing, as most people in its current form view it the way I've just explained it - it calls die() if the nonce is invalid. Yet here we are, with the nonce being invalid, and the function not terminating the execution. If you don't know about this quirk and you don't pass the $action on an admin script - you will introduce a CSRF vulnerability.

How do you protect yourself against that? Pass the bloody $action! Not passing the $action is obsolete since WordPress 3.2. If you specify the $action (the nonce's name), as is the only accepted modern practice, you are safe and you can ignore the last few paragraphs. Still, it's an interesting "deep dive" fun fact.

check_ajax_referer() is pretty similar to check_admin_referer(). It differs from the latter by checking for the name "_ajax_nonce" in $_REQUEST before checking for "_wpnonce" (assuming the name is not explicitly passed). It also dies when the nonce is invalid.

Unlike check_admin_referer(), this function has no backward compatibility quirk for checking the referer. It doesn't do anything with a referer at all, which makes its name another misnomer. It was probably named that way to keep consistency over correctness.

This function is supposed to be used when verifying nonces for AJAX requests. That's its semantic meaning, and that's where you should use it. We've not yet covered AJAX, so there's not much to talk about right now. I might or might not mention it again when discussing AJAX later.

Nonces For Non Logged-In Users

Think back to the way nonces are computed. The list of all variables used to calculate the hash is: action, tick, user ID, session token, and secret keys. The action, tick, and secret keys are the exact same for all users. What makes nonces unique for different users are the user ID and the token.

But what happens if a user is not logged in? Their user ID is 0 and the token is an empty string. Suddenly, all of the variables are the same for all logged out users. This is a fundamental problem with WordPress nonces. They do *not* protect anonymous users from CSRF attacks.

An attacker can simply scrape your website while logged out, and they will get the exact same nonce every other logged out user gets. This is why you should be very careful when adding actions on your website that do not require the user to be logged in.

Thankfully, that's not a typical case. I mean, what can an anonymous user really do? In most cases, the most "invasive" action they can take is send a publicly visible contact form. That's not a CSRF target. I can't really come up with any example of anonymous actions worth targeting with CSRF. Maybe if you tried to create some proprietary user sessions system outside of WordPress's API, but that's stupid and you should know that already.

Anyway, if you ever do find this behavior problematic, I have good news - you can change it. The filter "nonce_user_logged_out" is run for the user ID when it's 0. Using this filter, you can modify the user ID, making the nonces be different for logged out users.

Nonces With Caching

Nonces are perhaps the biggest pain in the ass when you try to introduce caching (especially full-page caching) on your website. Full-page caching is when you create a static HTML file for a page and serve it instead of running the PHP code on every request.

What happens if a cached page contains something with a nonce? Well, the nonce can get stale and the functionality will break. Let's assume you're using the default lifetime of a nonce - 24 hours. You have a form on the frontend that uses a nonce. You set up a caching plugin with no expiration of the cache. After (up to) 24 hours, your form will not work for any user, because the nonce on the page will not match the nonce recomputed on form submission.

How do you solve this? There are two answers to this question - the ideal way and the realistic way. We'll start with the latter.

In reality, most plugins and websites solve this by stepping over it. What I mean is: first, they don't include nonces for publicly available actions. That's the public contact form on the frontend from the previous section.

This form really doesn't need a nonce. What are you protecting the form against? A CSRF attack on a public form the attacker could just submit themselves? It doesn't need a nonce! The only real vulnerability is impersonating a sender by submitting the form using their IP.

The second part of this puzzle is disabling caching completely for logged in users. That's the "stepping over it" part. Most caching plugins, by default, do not serve the cached version to logged in users. This way, if the page uses a nonce, it will be computed on every request.

To be fair - not caching and/or serving cached pages to logged in users is not only due to nonces. It's a multi-factor problem with potential for user data poisoning the cache or leaking entirely. Some plugins allow you to keep a separate cache set for each user, but most still default to not caching logged in users at all.

This approach is not flawless. Imagine what would happen if you installed a badly written plugin that adds a form to the frontend. This form is completely public, but the plugin's author for some reason decided to use a nonce. Your caching plugin caches this page and serves the cache to all anonymous users. The nonce will eventually become stale and the functionality provided by the plugin will not work. Your only hope is setting the cache's expiration to the half-life of the nonce.

A more professional approach is fetching the nonce dynamically. You could create an AJAX/REST API endpoint on your website that returns a nonce. Then, instead of including the nonce in the markup generated by PHP, you just enqueue a JS script that retrieves the nonce from this endpoint and injects it into the HTML. This way, the cached HTML never sees the nonce.

This is an advanced technique, and I personally have not yet had the need to use it. I can't really say more about it than what I've already said.

I suppose there's a case to be made for a third, hybrid approach. If you had granular control over your cache rules and knew exactly which pages on your website did and did not contain

nonces, you could set your caching up so that only the pages without nonces get cached. That approach could work, and can even allow you to cache some pages for logged in users, but it's rather brittle and likely not good in 99% of cases.

Database Security ($wpdb)

We've already covered XSS (sanitizing and escaping) and CSRF (nonces). It's time we tackle the third type of vulnerabilities mentioned at the beginning of this chapter - SQL injections.

The first and most important rule of keeping your database secure against SQL injections is to use built-in WordPress functions if you can. Don't write a query to update a meta value, use update_post_meta(). Don't write a query to read an option, use get_option(). Don't write a query to get posts, use WP_Query. Most core WordPress functions are safe and handle sanitizing for you.

That being said, there are legitimate situations where you have to write your queries by hand, e.g., if you use custom tables or need maximum performance. To do that, you should use the global $wpdb object. It's an object of the wpdb class (yes, the name of the class is the same as the object).

wpdb is WordPress's abstraction over a database connection. It has many useful methods which we'll talk about soon. Every wpdb object (you can instantiate your own) is connected to exactly one database. The global $wpdb object is a connection to the main database configured in wp-config.php. By the way, this object is used internally by WP_Query to query the database.

Most methods in the wpdb class expect an SQL query as a string. Some of these methods are:

- **query()** - a general "run this query" method (returns the number of affected rows).
- **get_results()** - a general "run this query and return the results" method.
- **get_var()** - return just one value.
- **get_row()** - return a selected row from the query.
- and more…

You can hard-code these queries and they will be safe. The problem starts when you need to include user-provided data. You should remember the danger of that from our discussion on SQL injections. The user passing "abc;DROP TABLE wp_posts" would drop your posts table if you just concatenated the data with your query.

To make the query secure, you need to escape the untrusted data. In WordPress (and PHP), this process is called "preparing" the query. Here's how you do it with wpdb:

```
$prepared = $wpdb->prepare( "SELECT * FROM $wpdb->posts WHERE ID = %d", $post_id );
```

As you can see, you write the query using sprintf-like syntax (with placeholders, like %s, %d, and %f). You then pass the data you want escaped as arguments. They are then inserted in place of those placeholders.

The prepare() method makes sure the data is in the correct format and that it doesn't break out of the current query with a rogue semicolon. If you used %d but passed in 1.1 (a float), the escaped value will be 1 (an int) and will not be wrapped in quotes. If you use %s (a string), the value in the final query will be wrapped in quotes. All unsafe characters in the variables will be properly escaped. This helps ensure data validity and prevents SQL injections.

It's important to remember that prepare() returns the prepared query. In our case, the query is assigned to the $prepared variable. It's just a string you pass to the method you're calling. By the way, the $wpdb->posts property is the name of the posts table with the correct table prefix (configured when installing WordPress).

Sometimes you may need to create a query where you don't know the number of variables. Like a query with an IN clause and a user-provided list of IDs. Thankfully, prepare() lets you pass an array as the second argument. Note that, in this case, you have to dynamically create the query with a correct number of placeholders (equal to the number of values in your array) and pass this query as the first argument to prepare().

One more method you might find useful is **esc_like()**. It's used for escaping the contents passed to the LIKE clause. It's job is escaping the percentage sign (%) and underscore (%) so that they don't get interpreted as wildcards but as the content. In reality, this function just adds a backslash (\) before them. Use it on the value of the like (without the wildcards) before you pass it through prepare(), like so:

```
$search = '43% of planets'; // This might be what a user inputs in a search
$like = '%' . $wpdb->esc_like( $search ) . '%';
$sql = $wpdb->prepare( "SELECT * FROM $wpdb->posts WHERE post_title LIKE %s", $like );
```

$wpdb->esc_like($search) will return "43\% of planets". If you didn't do that, your actual LIKE clause would be: "%43% of planets%". See the problem? This query would return all posts whose post_title contains "43", not "43% of planets", because the first percentage sign is interpreted as a wildcard (assuming this query won't crash, which it probably will).

XML-RPC

XML-RPC is a **R**emote **P**rocedure **C**all protocol. It's basically just an API using XML payloads to trigger method calls on a remote server. It allows someone from the outside to prompt your website to do and return something (which is what pretty much all remote APIs do).

It's an old protocol, widely considered to be legacy. It was created in 1998, 2 years before the REST architecture. It was in WordPress since before its inception, as it was a part of the b2 system, which WordPress is a fork of. The protocol's main goal was why we use APIs now - to allow internet services to communicate with each other.

In wordpress, XML-RPC has been replaced with the more modern REST API. Right now, there's only a handful of popular services that use it, such as the WordPress mobile app or the Jetpack plugin. Unfortunately, this protocol and its implementation has proven very problematic when it comes to security, especially since it's been enabled by default since WordPress 3.5 (and still is!).

The XML-RPC endpoint in WordPress is /xmlrpc.php. This is the "front controller" for this API. There are a large number of methods available at your disposal - some innate to the protocol and some specific to WordPress. An example payload asking for a list of all supported XML-RPC methods looks like this:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>system.listMethods</methodName>
  <params></params>
</methodCall>
```

Historically, there have been a lot of security problems with this API. We're talking - privilege escalations, SQL injections, lack of capability checks, and more. It's also relatively common for plugins to have vulnerabilities exploitable only with XML-RPC. There are, however, two types of vulnerabilities you will always hear about when discussing this API in WordPress.

1. Brute Force Amplification Attacks

Many XML-RPC methods require authentication. This is achieved by passing the login and password in the XML payload. If the credentials are invalid, the server will return a 403 response.

This, in it of itself, can be considered a vulnerability, although a mild one. It provides an endpoint for the attacker to brute force passwords while skipping security checks (such as 2FA or captcha) implemented on the frontend login page.

But that's not what we're talking about here. We're talking about the system.multicall method. This protocol method allows the sender to include multiple (theoretically unlimited) methods in just one request. The idea was good - if you need to prompt multiple actions, it's more efficient to do it in one request. But the implications of that were a glaring oversight.

With system.multicall, you could try calling a method a thousand times in one request - each time with a slightly different password. One request would test 1,000 passwords. This is crème de la crème for brute forcing.

Thankfully, this is not a threat anymore. WordPress core developers understood that it was a big deal and changed their XML-RPC implementation (against the protocol's specification) so that, with system.multicall, if an authentication fails - all further authenticated methods fail as well. Their credentials are not verified. This change was introduced in WordPress 4.4.

2. Pingback DDoS Attacks

Another troublesome method in XML-RPC is pingback.ping. This method is supposed to notify a website that another website has linked to their blog post.

Here's how it works:

1. The owner of website A writes a blog post and links to a post from website B.
2. Website A sends a pingback.ping request to website B. The payload includes the source (a link to the new post on website A) and the receiver (a link to the post on website B that was linked to).
3. Website B receives this request and checks that website A is not lying. It sends a GET request to the source URL to see the linked URL with its own eyes.

All is fine until you realise that you can spoof step 2 without step 1. You can send a pingback request to any website which supports it. There's no validation that you actually are the source website (website A).

Here's how it can be used in DDoS attacks. An attacker gathers a huge list of websites that support XML-RPC and pingback.ping (which isn't going to be hard considering it's the default configuration). I'll call these sites "mules" from now on.

For each of the mules, they send an XML payload with the source being a link to their DDoS target, and the receiver being any link on the mule's website. The mule follows step 3 - it tries to verify that the link is actually there.

What this leads to is thousands of legitimate websites making GET requests to one server. Each of these "mules" (which are just normal WordPress websites) just became a part of a botnet DDoS'ing the target website. The target website will almost certainly go down, especially if the targeted URL is a large resource or results in heavy computation.

Best part? The real attacker's IP never even hit the target's server. It only contacted the mules, and it didn't have to send many requests to each of them either (maybe even as few as one). It's also hard to rate limit such an attack. All of the IPs are different, and they are usually legitimate servers that shouldn't be blocked.

Unfortunately, this vulnerability can be utilized to DDoS anybody - not only WordPress websites. The target doesn't have to support XML-RPC. It's the vulnerable "mules" that are causing harm to third party websites.

By the way, don't think that, just because you aren't the target, you should not care to not be the mule. Remember that each of these requests drain your server's bandwidth. Perhaps even more importantly, it can lead to your IP being blacklisted or even to violating the ToS of your hosting provider.

Disabling XML-RPC

Unless you or any of the plugins on your site are actively using XML-RPC, it's a best practice that you disable it completely. It's a legacy feature with many security holes and really no benefit.

There are a couple of ways you can disable XML-RPC on WordPress. The easiest one is with plugins. Many security plugins have built-in features allowing you to disable it with one click.

However, a better approach is to deny all requests to /xmlrpc.php at the web server level (e.g., in .htaccess). According to a WPScan article, many plugins only disable XML-RPC methods that require authentication, leaving other methods (like pingback.ping!) still working. That's because they (incorrectly) use the "xmlrpc_enabled" filter, which doesn't disable "anonymous" methods.

Keeping Your Site Updated

This is going to be a small section and I've already mentioned that in the security mindset, but it's so important that I couldn't not write about that. I wasn't joking when I said you should keep your code updated. Probably 98% of all WordPress "hacks" come from vulnerable themes or plugins that have not been updated by the website's maintainer.

Seriously, not updating your plugins is going to get you in trouble. Nobody said that owning a WordPress website is going to be effortless, and if they did - tell them they are idiots. Always make sure you plan for having a staging website to make updates on first (you don't update straight away in production, do you?). Update regularly, like at least once or twice a month.

Also, if your site is anything more than a worthless blog (and especially if you store sensitive personal data), get a good security plugin. There are many options: Wordfence, Sucuri, etc. They will notify you and add firewall rules if a vulnerability is discovered in any of the plugins on your website.

AJAX

AJAX stands for **A**synchronous **J**avaScript **A**nd **X**ML. AJAX is generally a pretty major point of confusion. First and foremost - it is not anything specific to WordPress. Let's talk about it more broadly first and then we'll cover WordPress's implementation.

I will start with what it's not. It is not a protocol, not an API, not a library, not a framework, and not any specific technology. AJAX is a set of techniques used to make a website update its content without a full page reload by asynchronous communication with the backend. That's it.

Whenever you have JavaScript on the frontend that sends a request to an endpoint and updates the DOM using the response content - that's AJAX. Most modern websites and web development frameworks use AJAX extensively. Client-Side Rendering and Single Page Applications are all AJAX.

Adding a product to cart without a page reload? AJAX. Autocompletion in Google's search bar? AJAX. Infinite scroll? AJAX. AJAX is everywhere, especially as web development progresses, because it improves the user experience.

Remember - it's just an abstract categorization term for an infinite amount of implementations. You may ask "how does it differ from a REST API?". Well, a REST API (or any other backend API) is a part of AJAX. It's the concrete thing that allows the client to communicate with the server, therefore allowing for AJAX to be used.

The X (for XML) is a legacy quirk. Back when AJAX was being developed (1999), XML was the standard format for sharing data between applications. Nowadays, most AJAX implementations rely on JSON (as most APIs use it).

admin-ajax.php

Most WordPress developers do not know the definition of AJAX. They confuse it with a specific implementation that's been in WordPress since 2006 - the admin-ajax.php file. This is what they usually think about when they say "use AJAX". I will refer to that implementation as "admin-ajax".

The admin-ajax.php file predates the REST API (which we will cover soon) by 10 years. For a long time, it was the only way (besides custom endpoints and using XML-RPC) to implement AJAX on WordPress websites. It's just one file in the wp-admin directory (the endpoint is /wp-admin/admin-ajax.php). It's the front controller for this implementation.

This file is surprisingly simple. It really boils down to 3 things:

- Loading the core of WordPress (wp-load.php).
- Checking for an action in the GET/POST data, i.e., $action = $_REQUEST['action'].

- Calling either do_action("wp_ajax_{$action}") or do_action("wp_ajax_nopriv_{$action}"), depending on if the user is logged in or not.

That's how this system works. You send a request in JavaScript to /wp-admin/admin-ajax.php with (at least) one argument - "action". This argument is just a string that you define as the name of your action. Then, WordPress calls do_action() for "wp_ajax_{$action}" if the user is logged in, or for "wp_ajax_nopriv_{$action"}" if the user is not logged in. Your job is to hook a handler callback to one or both of these actions, and to write and enqueue a JS script that makes the request on the client side.

Notice that all requests using admin-ajax are made to the same endpoint. The difference is only visible in the payload (the "action"). Therefore, if you installed a lot of plugins which utilize this API and opened the devtools network tab, you'd see a lot of seemingly similar requests, but they would all be used for widely different things.

Remember that for actions that do something (instead of only fetching data), you should use nonces to prevent CSRF. You should also not hard-code the /wp-admin/admin-ajax.php endpoint into your code unless you want your code to break all Wordpress installations running not from the root directory or with a renamed wp-admin folder. You'll see those principles in the example below.

Code Example

For this example, we'll develop a completely new plugin. This plugin will not only use admin-ajax, but will also utilize other concepts we've discussed in the last few chapters, such as user accounts, security, and internationalization. It will also follow the more modern single-class architecture. It will follow plugin development best practices (in contrast to our Post Reading Time plugin).

The plugin I'm talking about is a "Read Later" plugin. It will add a "Read this later" button to all blog posts. The user will be able to click this button and the post will be saved to their TO-READ list.

Here's the specification:

- Display a "Read this later" button at the top of post content for logged-in users.
- Allow the user to add (or delete) a post to their list without reloading the page (AJAX).
- Store the list as user metadata (in wp_usermeta).

For now, that's all. We will not display the user's list in any way. You can expand that if you want, but that's enough for demonstration purposes. I will also not style the button, as CSS is not the subject of this guide.

Let's start with the foundation - the class definition and the constructor:

```php
<?php
/*
* Plugin Name: Read Later List
*/

if ( !defined( 'ABSPATH' ) ) exit;

class Rltr_Read_Later {
    private int $user_id;
    private array $read_later_posts;

    public function __construct( int $user_id ) {
        // User is logged-out, get_user_meta() will return false.
        if ( $user_id <= 0) {
            $this->user_id = 0;
            $this->read_later_posts = [];
            return;
        }

        $this->user_id = $user_id;
        $this->read_later_posts = get_user_meta( $this->user_id, 'rltr_read_later');
    }
}
```

Nothing fancy here. Passing the user ID in the constructor is a good practice to make the class testable. We're storing the IDs of marked posts with an "rltr_read_later" meta key in the wp_usermeta table. It's not one entry with a serialized array - it's multiple entries with the same key (hence the third argument, "single", is left as default [false]). It's just a choice I made.

The rendering method:

```php
// Hooked to the the_content filter.
public function add_read_later_button( $content ) {
    if ( !is_singular( 'post' ) || !in_the_loop() || !is_main_query() ||
!is_user_logged_in() ) {
        return $content;
    }
    // Only show the button for logged-in users.
    if ( $this->user_id <= 0) {
        return $content;
    }
```

```
    $post_id = get_the_ID();

    // Check if the current post is already in the list to set the initial state.
    $is_saved = in_array( $post_id, $this->read_later_posts );
    $text = $is_saved
        ? __( 'Saved for later', 'read-later' )
        : __( 'Read this later', 'read-later' );

    $button_html = '<div><button class="rltr-read-later-button" data-post-id="' .
esc_attr( $post_id ) . '">' . esc_html( $text ) . '</button></div>';

    return $button_html . $content;
}
```

This method adds the toggle button right at the top of the post content. We're only showing it for single posts of type "post" and for logged in users. If the post is already saved, the button's content will be "Saved for later". Note the 'data-post-id' attribute. We'll use it in our JavaScript to pass the post ID in the admin-ajax request.

The backend admin-ajax handler:

```
// Hooked to the wp_ajax_rltr-toggle-read-later action.
public function ajax_toggle_read_later() {
    if ( !isset( $_POST[ 'post_id' ] ) || !is_numeric( $_POST[ 'post_id' ] ) ) {
        wp_send_json_error( 'post_id not set or not numeric.' );
    }
    $post_id = absint( $_POST[ 'post_id' ] );

    // This will check if the post exists and if it's of the supported post type.
    if ( get_post_type( $post_id ) !== 'post' ) {
        wp_send_json_error( "Post doesn't exist or its post type isn't 'post'" );
    }

    check_ajax_referer( "rltr-toggle-read-later-$post_id" );

    // Toggle the user meta.
    if ( in_array( $post_id, $this->read_later_posts ) ) {
        delete_user_meta( $this->user_id, 'rltr_read_later', $post_id );
        $status = 'removed';
    } else {
        add_user_meta( $this->user_id, 'rltr_read_later', $post_id );
        $status = 'added';
```

```
    }

    // Calls wp_die().
    wp_send_json_success( [ 'status' => $status ] );
}
```

Here comes the star of this chapter - the AJAX handler. Notice that we're utilizing validation (the security technique) twice at the top. First, we're checking if post_id is numeric - that's format detection. Then, we're checking if it's the ID of an actual post - that's a safelist. The next major security step is checking the nonce. We're only checking after and not before processing post_id because we need it for the nonce action.

Then, we either delete or add the post ID to the user's list, depending on if it's already in it (aka, toggle). wp_send_json_success() sends the response as JSON, but you could've just as well sent XML or even plain text (just remember to call wp_die(), otherwise your response will have a "0" appended at the end, which will break its structure).

admin-ajax handler for logged-out users:

```
// Hooked to the wp_ajax_nopriv_rltr-toogle-read-later-action.
public function ajax_handle_logged_out() {
    wp_send_json_error( 'You must be logged in to use this feature.' );
}
```

And here's the handler for the anonymous hook. This should never run for normal visitors, as the button is not rendered on the frontend for logged-out users.

Enqueue:

```
public function enqueue_scripts() {
    // Only load these scripts on single post pages.
    if ( !is_singular( 'post' ) ) {
        return;
    }

    wp_enqueue_script(
        'rltr-read-later-script',
        plugins_url( 'script.js', __FILE__ ),
        [],
        filemtime( plugin_dir_path( __FILE__ ) . 'script.js' ),
        true
```

```
    );

    $data_to_pass = [
        'ajax_url' => admin_url( 'admin-ajax.php' ),
        'nonce'=> wp_create_nonce( 'rltr-toggle-read-later-' . get_the_ID() ),
    ];
    // wp_json_encode ensures the data is safely converted to a JSON string.
    $inline_script = sprintf( "const rltr_ajax_obj = %s;",
wp_json_encode($data_to_pass) );
    wp_add_inline_script( 'rltr-read-later-script', $inline_script, 'before' );
}
```

And finally, a method used to enqueue our script file. Notice that we're passing the data from PHP to JavaScript using wp_add_inline_script() (which, as I already noted, is the correct function to do that). We're getting the admin-ajax URL using admin_url() and creating and passing a new nonce with the same action name as we're checking in the AJAX handler.

Setup (in the PHP file after the class definition):

```
function rltr_setup() {
    $rltr_read_later = new Rltr_Read_Later( get_current_user_id() );

    add_filter( 'the_content', [$rltr_read_later, 'add_read_later_button'] );
    add_action( 'wp_ajax_rltr-toggle-read-later', [$rltr_read_later,
'ajax_toggle_read_later'] );
    add_action( 'wp_ajax_nopriv_rltr-toggle-read-later', [$rltr_read_later,
'ajax_handle_logged_out'] );
    add_action( 'wp_enqueue_scripts', [$rltr_read_later, 'enqueue_scripts'] );
}
// Has to be on init because that's when the user is loaded.
// Doing setup when plugins are loaded wouldn't work because the user ID would
always be 0.
add_action( 'init', 'rltr_setup' );
```

Here's how we initialize the Rltr_Read_Later object and hook all the methods.

script.js:

```
document.addEventListener('click', function(e) {
    // Use event delegation to catch clicks on our button.
    if (!e.target.matches('.rltr-read-later-button')) {
```

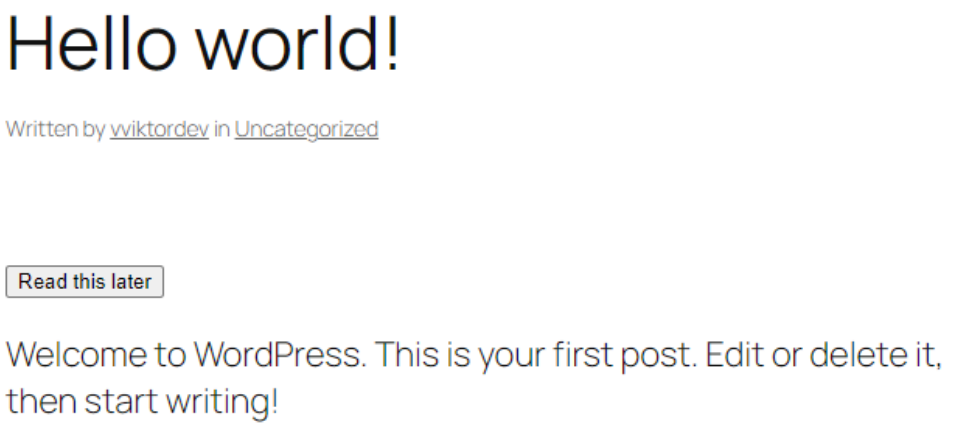
```
        return;
    }

    const button = e.target;
    const postId = button.dataset.postId;

    // Prepare the data to send.
    const formData = new FormData();
    formData.append('action', 'rltr-toggle-read-later');
    formData.append('_ajax_nonce', rltr_ajax_obj.nonce); // Nonce from inline script.
    formData.append('post_id', postId);

    // Disable the button to prevent multiple clicks.
    button.disabled = true;

    fetch(rltr_ajax_obj.ajax_url, { // URL from inline script.
        method: 'POST',
        body: formData
    })
        .then(response => response.json())
        .then(result => {
            if (result.success) {
                // Update button text and style based on the response from the server.
                if (result.data.status === 'added') {
                    button.textContent = 'Saved for later';
                } else {
                    button.textContent = 'Read this later';
                }
            } else {
                // If the request fails, show the error in the console.
                console.error('Error: ' + result.data);
                alert('Something went wrong. Please try again.');
            }
        })
        .catch(error => console.error('Network Error: ', error))
        .finally(() => {
            // Re-enable the button after the request is complete.
            button.disabled = false;
        });
});
```

Finally, the JavaScript. We're listening for all clicks of our button. When a click happens, we read the post ID from the data attribute and make a request using the Fetch API (you could've just as well used jQuery or the legacy XMLHttpRequest).

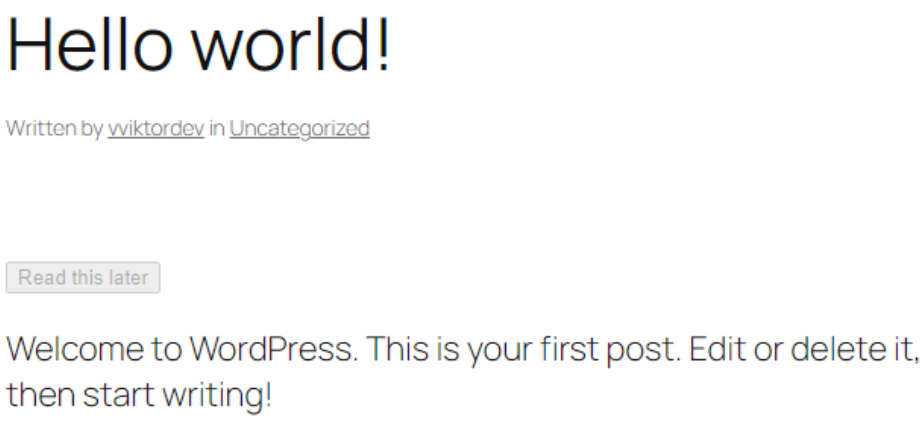
The request is a POST request with fields: "action" ($action in "admin_ajax_$action" hooks), "_ajax_nonce" (read from the passed inline JS), and "post_id". If the request succeeds, we change the button's content depending on whether the post was added to the list or deleted from it.

I would love to be able to embed a video here, but I'm writing it as a PDF and I'm not going to spend an hour trying to make that work. Here are three frames from using this feature.

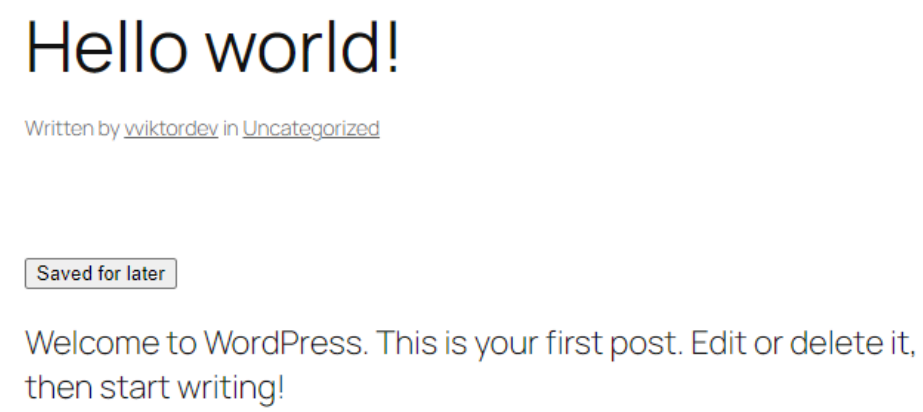
Before clicking the button:



Right after clicking the button (you could add some loading animation here, I just disabled the button):



After receiving a successful response from the server:



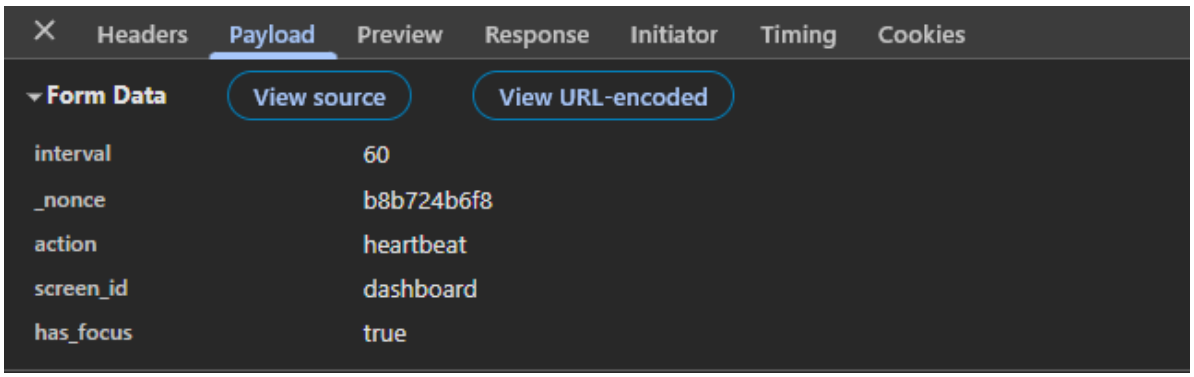
Heartbeat API

The Heartbeat API is a server polling API. Despite its scary name, the way it works is surprisingly simple. The entire idea is this: a small script is enqueued in the admin panel. This script sends an admin-ajax request every X seconds with a "heartbeat" action and some data. The server processes this request and returns a response that all scripts can see and act on. That's it.

But let's go deeper. First of all, we have to clear up what it is and why it exists. Imagine you had an ecommerce store. A cool feature in the admin dashboard would be to display a message in the bottom right corner "someone just made a purchase" every time someone buys something.

How would you implement it? Well, every 15 seconds you'd have to ask the server if a new order popped up. That's polling. This script would run all the time in the admin panel, and if a new order does indeed exist, it would add the HTML for the message. The Heartbeat API takes care of the first part (polling the server periodically).

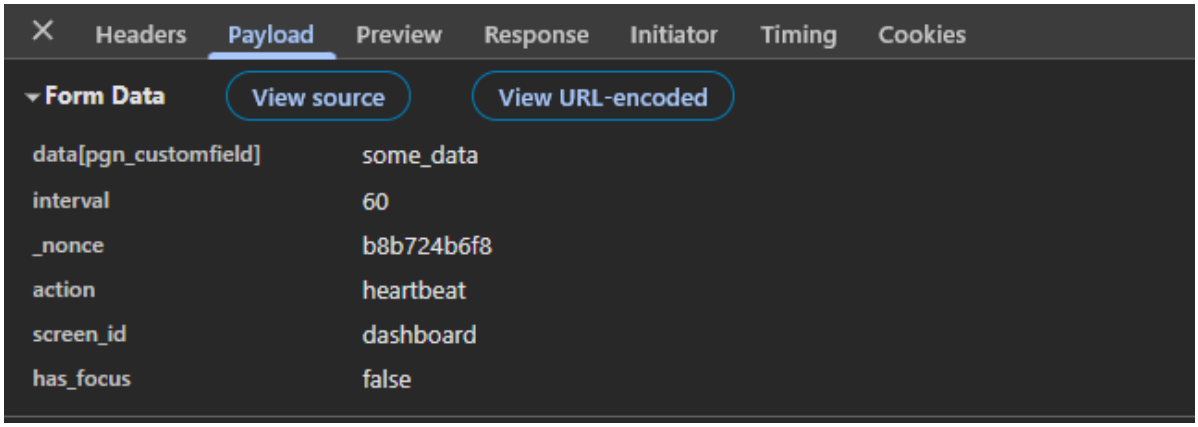
Here comes the "deep dive". On the frontend, this API is just a heartbeat.js file. It is only enqueued in /wp-admin/ by default (not on the frontend). Here's what a typical payload looks like (as I already said, the endpoint is /wp-admin/admin-ajax.php):



See the form data? We can add values here using jQuery. Yes, jQuery. This API was designed around it. It uses jQuery at its core (triggering jQuery events) and it's not possible to use it without it. Get over it. To add the data, we need to hook into the "hearbeat-send" jQuery event. Here's an example:

```
jQuery( document ).on( 'heartbeat-send', function ( event, data ) {
	data.pgn_customfield = 'some_data';
});
```

And here's the new admin-ajax payload:

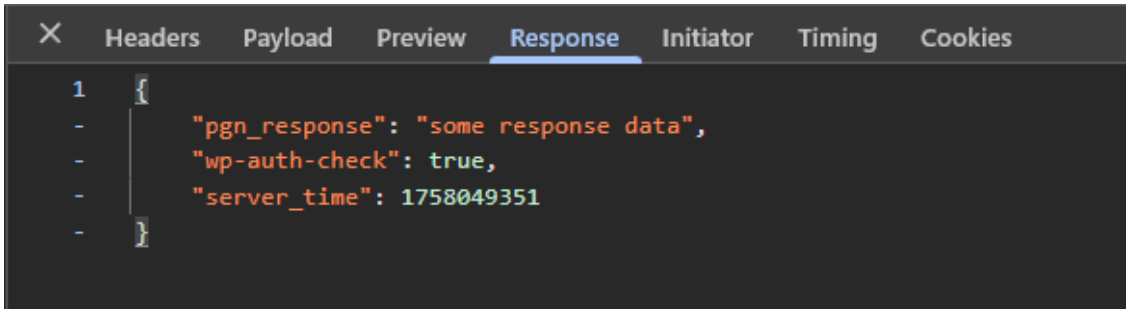


Great, we're now including custom data in every Heartbeat "tick". For our product alerts plugin, this could be as simple as including data.wooalerts_last_checked = {unix timestamp of last update}.

Remember, this request goes to the admin-ajax endpoint. However, the way you interact with that data on the server is a little different. You need to hook into the "heartbeat_received" filter. This filter passes the data (the payload from the request) and a $response array - which you can modify to change the response. Here's an example:

```
function pgn_receive_heartbeat( $response, $data ) {
	// If we didn't receive our data, don't send any back.
	if ( empty( $data['pgn_customfield'] ) ) {
		return $response;
	}

	$response['pgn_response'] = 'some response data';
	return $response;
}
add_filter( 'heartbeat_received', 'pgn_receive_heartbeat', 10, 2 );
```

And here's the response to the Heartbeat request:

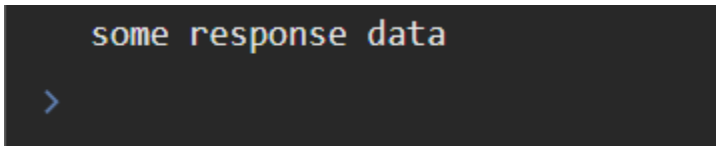


In reality, the semantically correct filter for response modification would be "heartbeat_send", but "heartbeat_received" (which is more intended for taking an action) will work just as well. It also passes a third argument - $screen_id. Look up at the screenshot of the payload. The screen ID was "dashboard". This is just a string indicating the admin page the user is on, e.g., "dashboard", "post", "plugins", etc.

The final piece of the puzzle is processing the response. In order to do that, you have to hook into the "heartbeat-tick" jQuery event. Let's say we'll log the response in the console every time we receive it:

```
jQuery( document ).on( 'heartbeat-tick', function ( event, data ) {
	if ( !data.pgn_response ) {
		return;
	}

	console.log( data.pgn_response );
});
```

And sure enough, here's our console after the next Heartbeat tick:



So, what is this API used for in the wild? Well, the core, as far as I can tell, uses it mainly for two things. First of all, it checks if your session is still valid. See the "wp-auth-check" field in the response above? If that returned false, you'd get a pop-up asking you to log in again. Go check it out for yourself - modify the session cookies in your browser while in the admin panel and wait for the next tick.

The second thing only exists in the post editor. There are a few custom data fields sent in the editor - wp-refresh-post-lock.lock and wp-refresh-post-lock.post_id. These fields are used to provide the "post locking" functionality. Basically, when you start editing a post, and another admin/editor wants to edit this same post, they will get a pop-up saying that the post is locked and being edited by you.

By the way, the polling in the editor happens every 10 seconds, not every 60 seconds like it does in all other parts of the admin dashboard. This API is very useful, but it can put stress on the server - especially if you have many users with access to /wp-admin/. That's why it's not by default enqueued on the frontend. Imagine 100 users making admin-ajax requests every minute. Most shared hostings would explode.

Thankfully, the API is smart. Look back up at the screenshot of the payload. It contains a "has_focus" boolean value. It indicates if the current window is in focus. If it isn't, the script will send only one final request (with a value of false) and it will not send any additional requests until the tab is back in focus. This means that you can live in peace with your 10 /wp-admin/ tabs open - they aren't sending 10 admin-ajax requests every minute.

However, if you still want to reduce the load the Heartbeat API has on your server, you very much can. The "heartbeat_settings" filter allows you to control the API's settings, including the interval. Here's how you could use it to set the interval to 60 for all admin pages (even the editor):

```
function pgn_control_heartbeat_interval( $settings ) {
    $settings['interval'] = 60;
    return $settings;
}
add_filter( 'heartbeat_settings', 'pgn_control_heartbeat_interval' );
```

REST API

REST stands for **RE**presentational **S**tate **T**ransfer. It's a system architecture developed with large distributed systems in mind (i.e., the internet). I will **not** teach you what REST is. This is not a topic for a WordPress deep dive. This is a topic for a system design deep dive (which I may do one day). As such, if you don't know what REST is, go and learn it on your own.

The WordPress REST API is WordPress's implementation of this architecture. It was added to WordPress in 2016 (version 4.7) and is the backbone of modern WordPress (particularly the Block Editor). It has only been gaining significance ever since. This chapter is extremely important if you want to be a competent WordPress developer - pay attention.

The Big Picture

Before we dive into how it works, we have to start with why it exists. The REST API is an attempt at decoupling WordPress from WordPress, however ridiculous that might sound. Before 4.7, WordPress was very self-contained.

What would you do if you wanted to create a mobile app for your website? Use the XML-RPC? You already know why that's a bad idea. Use admin-ajax? Good luck writing all the proprietary actions and defining their schema. There wasn't a widely accepted, standardized API allowing third party systems to interact with the data on your site.

The REST API changed that. First of all - REST was (and still is) by far the most popular architecture for web APIs. It uses JSON, which is a data format with excellent support across pretty much all languages. It is inherently decoupled from WordPress. Theoretically, no knowledge of WordPress is required to use the REST API - if you know REST then you know the rules of the API.

WordPress was created with the templating system in mind. This system, while good for simple blogs, doesn't scale well for enterprise-grade applications, which almost always require many external integrations.

The REST API provides a uniform way of accessing and modifying your site's data that is almost completely detached from WordPress. It is its own standardized and highly structured alternative layer you can use to interact with your website. You can use it to get a list of all posts, a single post's content, or even modify a post if you authorize (and all of that using simple JSON requests and responses).

Technical Details

The root URL of the API is /wp-json/. If you visited https://example.com/wp-json/, you'd see a list of all endpoints available on the site (spoiler: there's a lot). WordPress prevents name collisions with **namespaces**. The core namespace is "wp/v2". Our plugin's namespace could be "pgn/v1".

In the context of the WordPress REST API, a **route** is a URI (after /wp-json). The core posts route is /wp/v2/posts. An **endpoint** is a mapping of a route to an HTTP method. That is - "GET /wp/v2/posts" is an endpoint, and so is "POST /wp/v2/posts". The first one returns a list of posts and the second one lets you create a new post. They use the same route, but the endpoint (and the code they run) is completely different.

An example response to a GET request to /wp-json/wp/v2/posts/1/ is (truncated because it's large):

```
{
```

```
  "id": 1,
  "date": "2024-07-26T12:39:56",
  "date_gmt": "2024-07-26T12:39:56",
  "guid": {
    "rendered": "http://localhost:8001/?p=1"
  },
  "modified": "2025-09-16T21:14:08",
  "modified_gmt": "2025-09-16T19:14:08",
  "slug": "hello-world",
  "status": "publish",
  "type": "post",
  "link": "http://localhost:8001/hello-world/",
  "title": {
    "rendered": "Hello world!"
  },
  "content": {
    "rendered": "\n\u003Cp class=\"has-text-align-left\"\u003EHello,
World!abcdea\u003C/p\u003E\n\n\n\n\u003Cp\u003E\u003C/p\u003E\n",
    "protected": false
  },
  "excerpt": {
    "rendered": "\u003Cp\u003EHello, World!abcdea\u003C/p\u003E\n",
    "protected": false
  },
  "author": 1,
   // [...]
}
```


A list of all core endpoints, along with their schema definition (returned/expected fields and their data types), is available in the REST API Reference. Some of the more interesting routes include:

- /wp/v2/posts
- /wp/v2/pages
- /wp/v2/categories
- /wp/v2/comments
- /wp/v2/users
- /wp/v2/search
- and more…

The API supports GET query parameters like ?per_page=25&page=2 and many more. Different endpoints support different parameters and you can find them all in the reference. There are also global "meta-parameters" which can be used for all core endpoints. The most useful one is

"_fields", which lets you filter the fields returned, for example: /wp/v2/posts?_fields=id,author,title.

The Request Lifecycle

For the most part, the request lifecycle for a REST request looks exactly like a normal request (which we've covered at the beginning of this guide). It hits index.php, calls wp-load.php, then goes to wp-settings.php (which loads the entire core, themes, plugins, and calls the init action), and finally executes $wp->parse_request().

1. rest_api_init()

parse_request() is where a REST request diverges from a normal request, but we're getting ahead of ourselves. The first major step happens just before parse_request() runs - on the init hook (run at the end of wp-settings.php). The rest_api_init() function is attached to this hook. This function has only one job - to prepare WordPress for parsing the URL.

See, we've been using the pretty permalink version of the REST API, but if you remember from the beginning of this guide, that's not the default permalink structure. The default structure (plain permalinks) is dependent on query parameters, i.e., example.com?p=123. If you were to use the REST API with this structure, instead of accessing example.com/wp-json/wp/v2/posts, you'd have to access example.com/?rest_route=/wp/v2/posts. That's the plain equivalent of the pretty permalink.

WordPress core developers are smart. They knew it'd be a pain in the ass to try to handle different URL structures differently. Instead, why bother? The rest_api_init() function adds the ?rest_route parameter to the list of query vars and registers rewrite rules to move the value found after /wp-json/ to this parameter. Because of that, the internals of the API don't care if a pretty permalink structure was used. They always read the route from the "rest_route" query var. This is part of the Rewrite API which you'll learn soon.

2. URL Parsing

The parse_request() method is where this rewriting happens. It populates $wp->query_vars['rest_route'] with the route (e.g., "/wp/v2/posts"). At the end of its execution, it does a very important thing - it executes the parse_request action. This action is the last part the REST API request lifecycle shares with the lifecycle of a normal request. Remember, we're now in stage 6 out of 7 of a typical request lifecycle, just before querying the database for posts (which will never happen).

3. rest_api_loaded()

The function rest_api_loaded() is hooked to the parse_request action. The first thing it does is it checks if the request is a REST API request (by checking if the "rest_route" query var is not empty). If it is a REST request then we're officially in the "REST API request lifecycle" part. It defines the "REST_REQUEST" constant and continues handling the request.

The first step is instantiating a WP_REST_Server object. This object is responsible for handling the request, and that's exactly what happens when rest_api_loaded() calls $server->serve_request($route). This is the "front controller" for this class - it's the only method rest_api_loaded() calls before exiting.

But once again, we're getting ahead of ourselves. Right after instantiating the server, the "rest_api_init" action is fired (don't confuse it with the rest_api_init() function). It is on this action that you are supposed to register your endpoints. All of the core endpoints, along with custom ones from themes and plugins (which we'll cover soon), are registered now. The WP_REST_Server object stores all of these endpoints (their routes, methods, callbacks, etc.) in an array.

4. $server->serve_request()

As I already said, after registering all the endpoints, rest_api_loaded() calls the serve_request() method on the server. This is the main method that handles the request. It begins by instantiating a WP_REST_Request object. This object is just a one-stop object for all information about this request, i.e., the method, route, body, headers, parameters, etc. Most of its methods are getters and setters.

The next step is returning CORS headers and checking authentication (which we'll cover later). Then, assuming authentication passed, the dispatch($request) method is called. This method finds the correct endpoint handler (callback), validates and sanitizes the parameters, calls the permission callback, executes the handler (which usually returns an associative array that will eventually become the response body), and finally creates and returns a WP_REST_Response object.

Finally, other response headers are sent along with the status code and the response (encoded to JSON) is echoed.

5. die()

The execution goes back to rest_api_loaded() which calls die().

Authentication

Let's go back to the authentication check performed in the middle of $server->serve_request(). It's really just one line: apply_filters('rest_authentication_errors', null). All authentication functions hook to this filter. They have to return WP_Error if authentication fails, true if it succeeds, or null if it wasn't applicable. The dispatch() method is not called only if the value returned is a WP_Error.

In reality, when you think of "anonymous" requests to the REST API, you're thinking about these filters returning null. There are two authentication methods available by default: cookie

authentication and basic auth with application passwords. Both of their authentication checking functions hooked to this filter return WP_Error only if the authentication is actually attempted in the request but fails (a bad nonce, a wrong password, etc.).

If no authentication is attempted - they will just return null. But what if the endpoint requires authentication, like the endpoint for creating a post? Well, in reality, the rest_authentication_errors filter is not the main gate to your endpoints. It's only a bouncer on the outside (before any real processing happens) whose job is to block requests that try to authenticate but fail.

The real endpoint-specific permission check happens in the permission_callback specified when registering the endpoint. This callback is supposed to check if the current user has the necessary permissions to access this endpoint. That's where you'd check if the user can create a new post and return false if they can't. That will stop the execution before the main endpoint handler is called and return a 401/403.

Cookie Authentication

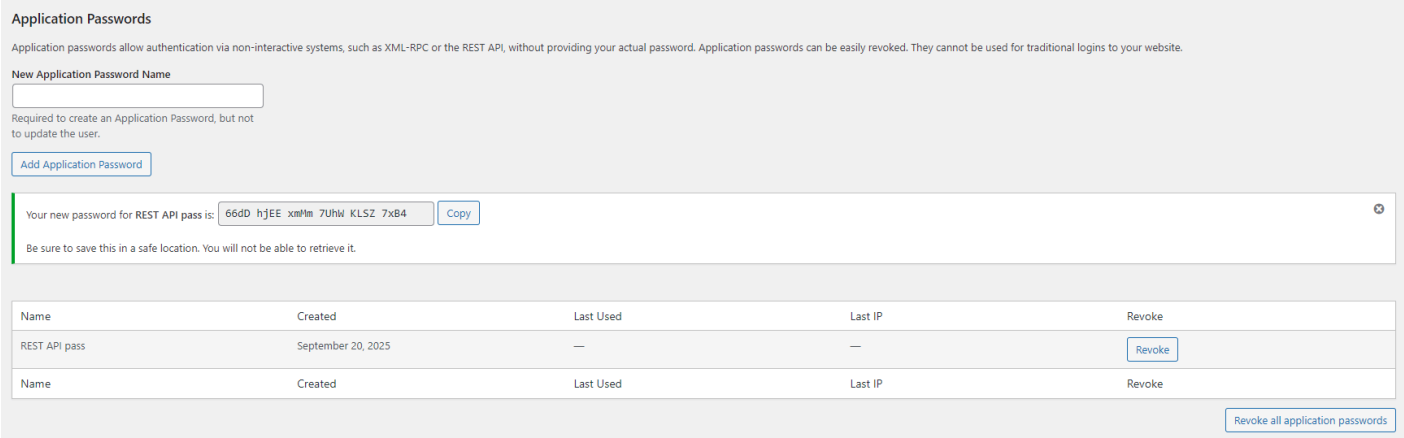
This is the standard authentication method used in WordPress. You already know everything about it. It works based off of cookies like wordpress_logged_in_{COOKIEHASH}.

Every request needs to contain a nonce - either passed as a _wpnonce GET/POST parameter or via the X-WP-Nonce header. The nonce uses the "wp_rest" action, that is - it's generated and verified with wp_create_nonce('wp_rest'). You have to pass this nonce to your script and include it in your requests (unless you're using the built-in WordPress Backbone.js library which takes care of that for you).

Notice that, because this authentication method is based on the authentication cookies and nonces, it really only works for requests made from the browser of a logged-in user. If you were to create a script that had to interact with the REST API, like a python script run from your computer, it wouldn't be practical to use this method. That's why application passwords were added.

Basic Authentication With Application Passwords

WordPress 5.6 added support for Application Passwords. An application password is a random 24-characters long alphanumeric token. Every user can generate multiple application passwords for themselves. An application password is always tied to a user account. Here's what adding an application password with name "REST API pass" looks like:



Application password can't be used for normal login with /wp-login.php. They are meant only for non-interactive systems, such as the REST API and XML-RPC (and other APIs added to WordPress in the future).

You can authenticate with the REST API using an application password with the Basic Auth (RFC 7617) scheme. This HTTP authentication method is, as the name implies, basic. It sends an Authorization header with a base64-encoded string containing the username and password (in the "username:password" format). Here's what such a header might look like:

```
Authorization: Basic ZGVlejpudXRz
```

Where "ZGVlejpudXRz" is the encoded username:password string. Of course, this authentication is completely insecure over unencrypted connections, as the base64 string can be read and decoded by anybody. That's why application passwords are, by default, disabled on all WordPress websites that don't use SSL.

The username is obviously the username of the account you want to authenticate (the same used with /wp-login.php). The password is the application password generated for that account. You can't use your main account password you use on /wp-login.php - the basic auth method works only with application passwords.

An important detail is that this form of authentication sets the user up the exact same way a cookie-based authentication does. This means that you can still call current_user_can() and all the other user-related functions. You don't have to know which authentication method was used. For a command-line request with curl, authentication is as simple as:

```
curl --user "USERNAME:PASSWORD" https://example.com/wp-json/wp/v2/users?context=edit
```

Other Authentication Methods

The two aforementioned authentication methods are the only methods supported in the core. There are, however, many plugins that add additional methods, such as JSON Web Tokens, OAuth 2.0, API keys, and more.

Custom Endpoints (Code Example)

Core endpoints are useful, but the real power of the REST API is in creating your own endpoints. We'll extend our Read Later plugin to expose endpoints allowing the user to get a full list of all post IDs in their list, as well as find out if an individual post is saved or not. This code example is just a demonstration. Read the Extending the REST API documentation to learn more details.

You already know that registering endpoints should happen on the "rest_api_init" hook. What you don't know is that there's a helper function you use to do it - register_rest_route(). In our case, we'll have to create a few new methods in our Rltr_Read_Later class. Let's start by registering the endpoint returning a list of all saved posts:

```
public function rest_register_routes() {
    register_rest_route( 'rltr/v1', '/saved', [
        'methods' => WP_REST_Server::READABLE, // this is just "GET"
        'callback' => [ $this, 'rest_get_saved' ],
        'permission_callback' => 'is_user_logged_in'
    ] );
}

public function rest_get_saved() {
    return $this->read_later_posts;
}

// ... down in rltr_setup()
add_action( 'rest_api_init', [ $rltr_read_later, 'rest_register_routes' ] );
```

That's it. If you now went to https://example.com/wp-json/rltr/v1/saved?_wpnonce={nonce}, you'd see a list of all post IDs you have in your list, like: ["1","3","7"]. Let's go through this code step by step. The first argument to register_rest_route() is the namespace. For us, it's "rltr/v1". The general best practice is slug/version (if you ever introduced breaking changes to the API, you could increment the version).

The second argument is the route. The third argument is an array of args. The WP_REST_Server::**READABLE** constant is equal to "GET" - it's just more API native to use the constants. There are other constants like:

- **CREATABLE** - POST,

- **EDITABLE** - POST, PUT, PATCH,
- **DELETABLE** - DELETE,
- **ALLMETHODS** - GET, POST, PUT, PATCH, DELETE.

'callback' is the main callback (handler) for the API. For us, it's the method that returns the array of post IDs in the list. 'permission_callback' is the callback we talked about extensively in the Authentication section. Notice that we're passing "is_user_logged_in". This function will return false if the request is anonymous. That's why we had to manually use the _wpnonce query parameter in the URL - otherwise, the cookie-based authentication wouldn't work. You should use "__return_true" if you want your endpoint to accept anonymous requests.

That was a very basic endpoint. Let's now add the more interesting one - checking if a post is saved or not by ID:

```
public function rest_register_routes() {
    // [...]
    register_rest_route( 'rltr/v1', '/saved/(?P<id>[\d]+)', [
        [
            'methods' => WP_REST_Server::READABLE,
            'callback' => [ $this, 'rest_get_saved_single' ],
            'permission_callback' => 'is_user_logged_in',
            'args' => [
                'id' => [
                    'required' => true,
                    'description' => __( 'The ID of the post', 'read-later' ),
                    'type' => 'integer',
                    'validate_callback' => [$this, 'rest_get_saved_single_validate'],
                    'sanitize_callback' => 'absint'
                ]
            ]
        ]
    ] );
}

public function rest_get_saved_single_validate( $value ) {
    if ( ! is_numeric( $value ) ) {
        return new WP_Error( 'rltr_invalid_arg', __( 'ID must be numeric.',
'read-later' ), [ 'status' => 400 ] );
    }

    if ( get_post_type( $value ) !== 'post' ) {
        $error_message = sprintf( __( "Post with ID %d doesn't exist or isn't a
'post'.", 'read-later' ), $value );
```

```
            return new WP_Error('rltr_post_not_found', $error_message, ['status' => 404]);
        }

        return true;
    }

    // Assumes $request['id'] is a number and a post with this ID exists.
    public function rest_get_saved_single( WP_REST_Request $request ) {
        $is_saved = in_array( $request['id'], $this->read_later_posts );
        return [ 'is_saved' => $is_saved ];
    }
```

This one is a little more advanced. First of all - look at the route. We're using regex to accept any positive integer after /saved/. We are also storing said integer as "id", which will then be passed and stored as an argument in the WP_REST_Request object.

There's a new entry in the array of options - "args". This is an array of associative arrays, where the keys are the names of arguments passed along with the request (such as our "id"). We're basically defining its schema: whether or not it's required, its description, and its type. There are also two extremely important parameters: validate_callback and sanitize_callback.

validate_callback runs first. Its job is to validate the argument and return an error (or false) if the data is not valid. This will kill the execution before the main callback is ever called. Look at our rest_get_saved_single_validate() method. We're returning an error if the ID is not numeric or if it's not the ID of a post.

sanitize_callback is exactly what you think. It runs after validate_callback. Its job is to sanitize the data before the main callback is executed. In this case, we're running our ID through absint.

By the way, be careful with what functions you use directly in these callbacks. They pass 3 parameters - $value, $request (current WP_REST_Request), and $key ("id" in our case). Some built-in PHP functions throw fatal errors if you pass them more arguments than they expect. One such example is is_numeric(), which is a very enticing function for validate_callback. In such a case, you'd have to wrap it in an anonymous function.

You can use functions like absint() without any problems, which is what we're doing with sanitize_callback. I'm not 100% sure why, but I think it has something to do with built-in PHP functions being written in C (and therefore being more strict), while userland functions are more lenient and automatically discard unused parameters. See ticket 34659 for a discussion on this topic (spoiler: it's dead).

Going back to the code example, we can see that our main callback is very simple - it checks if the passed ID is present in the list of all saved posts and returns a boolean value with the

"is_saved" key. We don't have to do any validation or sanitization inside of this method as it has already been done by the specific callbacks. By the way, you can treat the WP_REST_Request object like an array because it implements the Array Access interface.

Here's an example response for https://example.com/wp-json/rltr/v1/saved/1?_wpnonce={nonce}:

```
{"is_saved":true}
```

And here's one for https://example.com/wp-json/rltr/v1/saved/420?_wpnonce={nonce}:

```
{
  "code": "rest_invalid_param",
  "message": "Invalid parameter(s): id",
  "data": {
    "status": 400,
    "params": {
      "id": "Post with ID 420 doesn't exist or isn't a 'post'."
    },
    "details": {
      "id": {
        "code": "rltr_post_not_found",
        "message": "Post with ID 420 doesn't exist or isn't a 'post'.",
        "data": {
          "status": 404
        }
      }
    }
  }
}
```

Why The Previous Example Is Wrong

You read that right. The code example you just saw is theoretically wrong. It's not *wrong* wrong - if you navigate to the registered endpoints, you will get the data you expect. It's just that it doesn't really respect the REST architecture, which is something that might've been bugging you if you know a thing or two about REST.

A REST route should point to a resource. A noun - something you can touch (figuratively). A list of saved posts of a user isn't really a resource. It's a property of a resource - the user. That's consistent with the fact that we are storing this data as the user's metadata (in wp_usermeta).

This fact is actually very important. Because it's a property of a user, it should be available in the user resource. Guess what? It is. Or at least - it can be. The Modifying Responses documentation explains how to make metadata of an object (e.g., post, user, and other objects with _postmeta tables) become available in REST API responses. The trick is registering the metadata key with register_meta(). Let's add this code to our Rltr_Read_Later constructor:

```
register_meta( 'user', 'rltr_read_later', [
    'type' => 'integer',
    'single' => false,
    'show_in_rest' => true,
] );
```

I've already mentioned the register_meta() function in the past, but it was a long time ago. It was a prerequisite for using metadata with the Block Bindings API and having metadata revisioned. Anyway, if we add this piece of code, the response of GET /wp/v2/users/me?_wpnonce={none} will be something like:

```
{
  "id": 2,
  "name": "Admin nicename",
  // [...] truncated for brevity
  "meta": {
    "rltr_read_later": [1, 3, 7]
  },
  // [...] truncated for brevity
}
```

Let's be honest - the endpoint for checking if a single post is saved or not is pragmatically useless. Just get the list of all saved posts and check if your ID is in the array. It was for demonstration purposes only. That is the canonically correct way to implement support for the REST API.

PS: The "Modifying Responses" documentation also talks about register_rest_field(). It's a function allowing you to add/modify fields in responses of existing REST API endpoints. It's pretty advanced and niche so I'm not going to explain it here.

Controller Classes

Our example was simple. That's why we could hack together a working solution with a few methods. Now imagine doing that with an ecommerce plugin. If you didn't spend some serious time thinking about how to structure and name your callbacks, it would quickly become a hell to maintain.

Thankfully, you don't have to do that all by yourself. WordPress comes with an abstract WP_REST_Controller class. All controllers handling core wp/v2 routes (posts, users, etc.) extend this class. It contains a number of useful methods all core APIs support, such as get_fields_for_response(), get_context_param(), and more.

But that class doesn't solve the structure problem by itself. If you want your code to be easy to maintain, you should follow patterns set by the core. The best way to learn them is by reading the code. All core controller classes have very similar methods: register_routes(), get_items(), get_item(), get_item_permissions_check(), etc.

The point is - the code responsible for handling REST API endpoints is not trivial. If you're writing an advanced plugin, you should seriously consider its structure. The use of controller classes following the same patterns as the core will likely make it a much more pleasant experience. Read more about them in the official Controller Classes documentation.

Custom Post Types In The REST API

Remember the custom post type "thm_book" we created a while ago? We can expose it in the REST API. All we have to do is pass 'show_in_rest' => true in args. It will use the same controller as the core posts endpoint. You can do the exact same thing with custom taxonomies, like our "thm_genre" taxonomy. Read the docs for more information.

Headless WordPress

A typical website is usually divided into two parts - body and head. Body is the backend, while head is the frontend. The concept of a headless CMS is not new. There are many headless CMSes on the market. They provide you with a backend for creating your content and expose it via APIs. You are then only responsible for creating the frontend by reading this content.

That's exactly what headless WordPress is. It's a WordPress setup where WordPress doesn't render any HTML. The theme layer never runs. You use WordPress only for creating and storing posts (and other content) and your frontend is a separate application - usually using a modern JS framework like React, Vue, Svelte, etc. The content writers still log into /wp-admin, but the requests from users don't load the WordPress templating system.

The REST API is what makes this possible (although GraphQL can be used with a plugin). All the data on your site is available in the API. Remember when I talked about decoupling WordPress from WordPress? That's exactly what I meant - treating WordPress as only the backend.

Headless WordPress is a rather advanced architecture and you won't see it used often (nor should you). The data is either fetched from the API on the frontend (using Client-Side Rendering, usually with Single Page Applications) or a static HTML file is pre-generated (Server-Site Rendering with Static Site Generation).

Pros

- **Frontend freedom** - you are not tied to the WordPress way of creating frontend (particularly PHP templates). You can potentially use more modern practices and frameworks without having to fight the system.
- **Omnichannel publishing** - the same WordPress backend can be used to power your website, mobile app, desktop app, etc.
- **Scalability** - in big organizations, separate dev teams can work on the backend and on the frontend.
- **Security** - the presentation layer is completely detached from the backend, /wp-admin isn't on the same domain, and the number of plugins is much lower, usually reducing the number of attack vectors. This doesn't reduce the importance of keeping WordPress secured.

Cons

- **Complexity** - you are now creating and maintaining two separate applications - your JS frontend and your WordPress backend. This means two deploy pipelines, two hosting environments, etc.
- **Loss of themes and plugins** - most plugins are not headless-compatible. They interact with the presentation layer, often enqueueing scripts or filtering the_content (like our plugins did). You will have to write most of the custom backend functionality yourself.
- **More dev work (and costs)** - referring to the previous point, more functionality to write means more dev work, time, and budget needed to ship the project. You often have to reinvent things that would be easy to implement in WordPress.
- **Degraded editor experience** - you lose the live preview, the customizer, WYSIWYG editing, etc. (the theme is no longer what the frontend actually looks like).
- **SEO challenges** - if using Client-Side Rendering.

Overall, headless WordPress is an overkill for 99.9% of projects. Don't blindly fall for claims that it will make your site blazingly fast. This claim is based on the assumption that the site will be generated on the server and a static HTML file will be served. The problem is - you can do that with WordPress and page caching. In practice, speed isn't a deciding factor.

What is a deciding factor is the project's scope. If you're creating a complex web application with cutting-edge custom frontend requirements and a large team of developers, you might want to seriously consider going headless (or just not using WordPress at all). Otherwise, it's a waste of resources, especially since modern WordPress with Block Themes has really been closing the gap between modern web development (think back to the Interactivity API).

HTTP API

The REST API was about allowing inbound connections (third parties connecting with your server). The HTTP API is about outbound connections - your server sending HTTP requests to

third parties. It's the official WordPress way of sending HTTP requests. You don't use cURL or file_get_contents(). You use this API.

It's a rather simple API so we shouldn't spend as much time here as we did on the REST API. We'll cover making requests for every method and then we'll discuss accessing responses.

GET

To GET data from an external service, you should use the **wp_remote_get($url, $args)** function. $args is just an array of arguments. A full list of accepted keys and values is available in the WP_Http::request() documentation, but some of the most important for GET are:

- **timeout** - how long to wait before aborting the connection. Default 5.
- **redirection** - how many times to follow redirections. Default 5.
- **httpversion** - either "1.0" or "1.1". Default "1.0".
- **user-agent** - User-agent value sent. Default: 'WordPress/' . get_bloginfo('version') . '; ' . get_bloginfo('url').
- **blocking** - if false, the request will be sent asynchronously (PHP execution will continue without waiting for the response - use only if your code doesn't need the response). Default true.
- **headers** - array or string of headers to send with the request.
- **cookies** - array of cookies to send with the request.

Oh, and if you need to include query parameters in the URL, especially if they are included dynamically (e.g., read from the database), you will find the add_query_arg() function very useful. It's a generic WordPress helper for adding query params to URLs. Sending a GET request in practice is as simple as doing:

```
$url = add_query_arg( 'key', 'value', 'https://example.com' );
$response = wp_remote_get( $url );
```

POST

POSTing data is almost as easy as GETting data. You use the **wp_remote_post($url, $args)** function. There's one $args key I've deliberately not mentioned when discussing GET (because it shouldn't be used with GET requests) - '**body**'. This key accepts either a string or an associative array and defaults to null. This will be our payload. An example of a post request:

```
$args = [
    'body' => [
        'email' => 'mail@example.com',
        'message' => 'I hope you read the security chapter!;DROP TABLE wp_posts;'
    ]
```

```
];
wp_remote_post( 'https://example.com/contact', $args );
```

If you need to pass the data as JSON, you will have to encode it yourself using wp_encode_json() and pass it as a string to the 'body' argument. Remember to also set the content-type header to "application/json".

Unfortunately, it's not clear to me what the content-type is by default. The documentation doesn't mention it and I couldn't find it in code. I'm assuming it's application/x-www-form-urlencoded if you're passing an array. Keep it in mind if you have problems with POST requests and maybe set it directly or do some more testing.

Other Methods

All of the request methods actually use WP_Http::request() underneath. That's why their arguments are the same. One of the most crucial keys you can pass into $args that I've not yet mentioned is '**method**'. Although you could use it with the _get and _post helpers (which would be very confusing), it's more correct to include it when using the more generic **wp_remote_request($url, $args)** function.

The list of accepted 'method' values:

- 'GET'
- 'POST'
- 'HEAD'
- 'PUT'
- 'DELETE'
- 'TRACE'
- 'OPTIONS'
- 'PATCH'

Using this generic method, along with all the other options available to you in the $args parameter, you can send literally any HTTP request. Oh, and there's also a helper function for sending HEAD requests that might come in handy - **wp_remote_head($url, $args)**.

Responses

Because all the functions use the same underlying method for making the requests, their response structure looks exactly the same. This is convenient because I can cover responses only once instead of having to discuss them for every method individually. I made a GET request with wp_remote_get() to https://jsonplaceholder.typicode.com/todos/1 and printed the $response with print_r(). Here's what it looks like (truncated):

```
Array
(
    [headers] => WpOrg\Requests\Utility\CaseInsensitiveDictionary Object
        (
            [data:protected] => Array
                (
                    [date] => Tue, 23 Sep 2025 17:52:09 GMT
                    [content-type] => application/json; charset=utf-8
                    [server] => cloudflare
              [... truncated headers ...]
                )

        )
    [body] => {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
}
    [response] => Array
        (
            [code] => 200
            [message] => OK
        )

    [cookies] => Array
        (
           [0] => WP_Http_Cookie Object
                (
                    [name] => example_cookie_i_set_with_$args
                    [value] => example_value
                    [expires] =>
                    [path] => /
                    [domain] =>
                    [port] =>
                    [host_only] => 1
                )
        )

    [filename] =>
    [http_response] => WP_HTTP_Requests_Response Object
        (
            [data] =>
```

```
            [headers] =>
            [status] =>
            [response:protected] => WpOrg\Requests\Response Object
                (
                    [body] => {
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
                    [raw] => HTTP/1.1 200 OK
Date: Tue, 23 Sep 2025 17:52:09 GMT
Content-Type: application/json; charset=utf-8
Transfer-Encoding: chunked
Connection: close
Server: cloudflare
[... truncated headers ...]

{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
                    [headers] => WpOrg\Requests\Response\Headers Object
                        (
                            [data:protected] => Array
                                (
                                    [date] => Array
                                        (
                                            [0] => Tue, 23 Sep 2025 17:52:09 GMT
                                        )
                                    [content-type] => Array
                                        (
                                            [0] => application/json; charset=utf-8
                                        )
                                    [server] => Array
                                        (
                                            [0] => cloudflare
                                        )
                                   [... truncated headers ...]
                                )
```

```
                        )

                    [status_code] => 200
                    [protocol_version] => 1.1
                    [success] => 1
                    [redirects] => 0
                    [url] => https://jsonplaceholder.typicode.com/todos/1
                    [history] => Array
                        (
                        )

                    [cookies] => WpOrg\Requests\Cookie\Jar Object
                        (
                            [cookies:protected] => Array
                                (
                                    [example_cookie_i_set_with_$args] =>
WpOrg\Requests\Cookie Object
                                        (
                                            [name] =>
example_cookie_i_set_with_$args
                                            [value] => example_value
                                            [attributes] => Array
                                                (
                                                )

                                            [flags] => Array
                                                (
                                                    [creation] => 1758650522
                                                    [last-access] => 1758650522
                                                    [persistent] =>
                                                    [host-only] => 1
                                                )

                                            [reference_time] => 1758650522
                                        )
                                )
                        )
                )
            [filename:protected] =>
        )
)
```

As you can see, the response is actually just a big array. There are different data types inside of this array, such as CaseInsensitiveDictionary, WP_Http_Cookie, WP_HTTP_Requests_Response, and more. But why am I showing you this? So that you understand why we don't just read the data directly from this array and why you don't have to remember its structure.

You read data from the response array by using wp_remote_retrieve_*() functions. These are helper functions designed to interact with the structure of this array and return only the data you care about in a predictable format.

Here's a list of these functions. Check out their individual documentations when you need to use them:

- wp_remote_retrieve_body()
- wp_remote_retrieve_cookie()
- wp_remote_retrieve_cookies()
- wp_remote_retrieve_cookie_value()
- wp_remote_retrieve_header()
- wp_remote_retrieve_headers()
- wp_remote_retrieve_response_code()
- wp_remote_retrieve_response_message()

The response array is of course only returned if the request succeeds. But what if it fails? What if the timeout is exceeded? Or the server's network is so overloaded it starts rejecting requests at the TCP level? In those cases, the response is a WP_Error object. That's why you should always check if the response is an error before proceeding, like this:

```
$response = wp_remote_get( 'https://example.com/contact' );
if ( is_wp_error( $response ) ) {
    // handle error
} else {
    // proceed as if successful
}
```

Remember that WP_Error is only returned when there was an actual error with getting the response. If the response was received successfully but the status code was 4xx or 5xx, that would **not** return a WP_Error but a normal response array.

A nice touch of the HTTP API is that all wp_remote_retrieve_*() functions work even if you pass a WP_Error as the argument, i.e., they expect either an array or a WP_Error. Of course their behavior will be different if passed an error object (usually returning an empty string).

Security

There are two $args entries that relate to security. You should pay some attention to them, which is why I'm writing this section.

The first one is '**reject_unsafe_urls**'. It expects a boolean value. It's false by default, but if you set it to true, the URL of the request will be passed through the wp_http_validate_url() function. This function will abort the request if the URL is deemed unsafe. Some examples of unsafe URLs are:

- **ftp://example.com/caniload.php** – Invalid protocol – only http and https are allowed.
- **http:///example.com/caniload.php** – Malformed URL.
- **http://user:pass@example.com/caniload.php** – Login information.
- **http://example.invalid/caniload.php** – Invalid hostname, as the IP cannot be looked up in DNS.
- **http://192.168.0.1/caniload.php** – IPs from LAN networks.
- **http://198.143.164.252:81/caniload.php** – By default, only 80, 443, and 8080 ports are allowed.

This function is explicitly designed to secure you from Server-Side Request Forgery (SSRF) attacks. It's like CSRF but for servers (an attacker tricks your server into performing an action you didn't mean to perform).

You might be wondering if you need this if you're hard-coding the URLs. The answer is yes, and I hope to explain why in just one question: are you 100% sure the URL you hard-coded is not going to redirect (status code 3xx) to a different URL? I can answer that for you - no, you are not. with 'reject_unsafe_urls', not only is the first URL validated, but every redirect hop is validated as well.

As a matter of fact, this is such a big deal that there's an entire family of helper functions designed just so that they set this argument to true by default. That's the only way they differ from their aforementioned counterparts. These are:

- **wp_safe_remote_get()**
- **wp_safe_remote_post()**
- **wp_safe_remote_head()**
- **wp_safe_remote_request()**

The other important security argument in $args is '**sslverify**'. Thankfully, it defaults to true. If set and the request is sent using HTTPS, the SSL certificate will be verified against a bundled CA Root Certificates file (which you can change with the 'sslcertificates' argument by the way). If the certificate of the recipient fails verification, a WP_Error will be returned.

Hooks

This is one of the biggest reasons you should use the HTTP API instead of raw-dogging cURL. As with everything in WordPress, the functions and methods in the HTTP API provide many actions and filters allowing for seamless extensions. I'm not going to list all the hooks now - you have the documentation and the code for that.

That being said, some notable hooks include "http_request_args", "pre_http_request", and "http_api_debug". The first one allows you to filter $args for every request, which makes it easy to change the user agent or enforce 'reject_unsafe_urls' (or anything else your soul wants to do). "pre_http_request" filters the request before it is sent. This, coupled with "http_api_debug" which runs after the request, allows you to introduce caching of responses.

Rewrite API

This is a fascinating API. It is so fundamental to how WordPress works, yet so few people actually understand it. You are about to learn in detail how WordPress's pretty permalinks work and how you can create your own custom URL structures. Hopefully, this knowledge will elevate your understanding of WordPress internals, which, as you might remember, is the entire point of this guide.

Remember what I said about parsing the URL when I was discussing the REST API? I said that core WordPress developers were smart and instead of handling pretty permalinks differently than plain permalinks, they just "moved" the route to the "?rest_route=" query parameter and treated it like a plain permalink. Well, it turns out that's the way the entire rewrite system works, including all your pretty URLs to posts and pages.

add_rewrite_rule()

To explain it, I'll start by introducing the single most important function in the entire Rewrite API: add_rewrite_rule(). This function adds a regex rewrite rule. Here's how you could use it:

```
add_rewrite_rule('^/post/(.*)?', 'index.php?post_type=post&name=$matches[1]', 'top');
```

Adding this rewrite rule will make it so that visiting example.com/post/post-name will be equivalent to visiting example.com/index.php?post_type=post&name=post-name. This is how all pretty permalinks work in WordPress. When you set the permalink structure in settings to anything else than plain, all you're doing is triggering a set of those rewrite rules to be added. The internals of WordPress operate on the plain structure (with query string parameters) only.

The third parameter - "top", makes it so that this rewrite rule gets added at the top of the list of all registered rewrite rules. Rewrite rules are checked from the top to the bottom, and the first

one matching wins. It's to ensure that our custom rule isn't beat by some core rule. The other option is "bottom".

Query Variables

The path is always index.php, as that's WordPress's front controller. But what are those query parameters, like "post_type" and "name"? Well, their name in the WordPress world is "query variables". The global $wp object has a $query_vars array. This array is populated with all query vars found while parsing the rewritten URL. After parsing the previous example URL, this array would be equal to:

```
array( 'post_type' => 'post', 'name' => 'post-name' );
```

The reason it works this way is actually pretty simple. Remember what happens after the URL is parsed? The main query is run with WP_Query. Now, instead of having to do god knows what to construct the query, all WordPress has to do is read from this array, like this (conceptually):

```
$args = [
    // [...]
    'post_type' => $query_vars['post_type']
    'name' => $query_vars['name']
    // [...]
]
new WP_Query( $args );
```

Do you see how brilliant this is? We've translated a pretty URL that's hard for computers to interpret (/post/post-name) into a URL that's ugly to humans but is basically just a list of key-value pairs (?post_type=post&name=post-name). Then, we just use these keys and values (query vars) in all further processing of the request, not having to worry in the slightest about what the original URL looked like. That's the power of the Rewrite API.

This is the general idea, but query vars have some more complexity tied to them. First, not all query parameters will become query variables. The $wp object has another array - $public_query_vars. This array is basically a whitelist of all allowed query vars. It's long, but some of the most important ones include:

- **p** (post id)
- **category_name**
- **search**
- **name**
- **author_name**
- **page_id**
- **post_type**

- and more…

If you visited example.com/?arbitrary=abc, there would be no $query_var['arbitrary'], as the key "arbitrary" is not whitelisted. That being said, "rest_route" is not included in the default whitelist, so how does it work? Meet **$wp->add_query_var()**. This method allows you to add a custom query variable to the list of all allowed query vars.

You should now know enough to be able to understand the code snippet responsible for making the REST API's permalink structure work. Here it is (a little simplified):

```
global $wp;
$wp->add_query_var( 'rest_route' );
add_rewrite_rule( '^' . rest_get_url_prefix() . '/?$', 'index.php?rest_route=/',
'top' );
add_rewrite_rule( '^' . rest_get_url_prefix() . '/(.*)?',
'index.php?rest_route=/$matches[1]', 'top' );
```

What this does is it registers "rest_route" as a whitelisted query variable and then adds rewrite rules to translate the pretty URLs to the plain URLs. The root REST API URL (example.com/wp-json/) is translated to ?rest_route=/, and if there's a route (e.g., example.com/wp-json/wp/v2/posts), it becomes the value of rest_route (i.e., example.com/index.php?rest_route=/wp/v2/posts).

But what does the REST API do with this since it doesn't run WP_Query? Well, remember the rest_api_loaded() function? Here's how it uses it to check if the request is for the REST API:

```
if ( empty( $GLOBALS['wp']->query_vars['rest_route'] ) ) {
	return;
}
```

And here's how it then reads the route to pass it to $server->serve_request():

```
$route = untrailingslashit( $GLOBALS['wp']->query_vars['rest_route'] );
if ( empty( $route ) ) {
	$route = '/';
}
$server->serve_request( $route );
```

That shows perfectly that it's entirely up to you as to how you use query vars. Some query vars, or maybe even all of the default ones, are used when constructing the main WP_Query. But that's not a rule. You can register new rewrite rules, capture the values from the URL to your own query vars, and use them to return any arbitrary content (like the REST API does).

PS: You can also whitelist custom query variables using the "query_vars" filter.

Parsing The URL Step By Step

I want you to *really* understand it, which is why I'm going to go even closer to the actual implementation. All of what we're talking about happens in WP::parse_request() (stage 6 of the request lifecycle). Here's an ordered list of things done by this function:

1. Fetch all registered rewrite rules (from $wp_rewrite, which is a global instance of WP_Rewrite).
2. Prepare the URL path ("https://example.com/post/post-name" becomes "post/post-name")
3. Iterate over all rewrite rules and try to match them with the path using preg_match(). The first matching rule is saved and breaks out of the loop.
4. Convert the path ("post/post-name") into the plain path defined in the matched rule (index.php?post_type=post&name=post-name).
5. Run this plain path through parse_str() to get an array of query parameters (i.e,. array('post_type' => 'post', 'name' => 'post-name')).
6. Iterate through $public_query_vars (the list of all allowed query vars) and try to find and add them to $query_vars from one of these, in order:
    a. $_POST
    b. $_GET
    c. The array of query parameters we just created with the rewrite rule.

That's it. After this method runs, we will have our $query_vars array populated with all query variables present in the parsed plain URL, which was, in turn, created from the pretty URL using rewrite rules we've registered.

You might've noticed something interesting. $_POST and $_GET are checked before the array generated with rewrite rules. This has some interesting consequences. As an example, take this URL: example.com/wp-json/wp/v2/posts/1?rest_route=/wp/v2/posts/2. What post ID will it return data for - 1 or 2? The answer is 2.

This is also why example.com?p=123 will always render the page for post with ID 123, even if pretty permalinks are enabled (although it will redirect you to the appropriate pretty URL), whereas example.com/post-title-of-post-123 will throw an error as soon as pretty permalinks are switched off. Query variables passed as if the permalinks structure was plain will always work.

Example

Let's imagine you were creating a website for a library. Every book in the library was placed on a shelf. You wanted to allow visitors to display all books on a given shelf by navigating to example.com/shelf/{shelf_id}, where shelf_id is something like "a", "b", etc. Let's also imagine

that, for some reason, the list of all books stored on a given shelf is kept in a custom database table. Here's how you'd manage adding that URL structure:

```
function pgn_configure_shelf_rewrite() {
    global $wp;
    $wp->add_query_var( 'shelf' );
    add_rewrite_rule( 'shelf/(.*)?', 'index.php?shelf=$matches[1]', 'top' );
}
add_action( 'init', 'pgn_configure_shelf_rewrite' );

function pgn_filter_posts_by_shelf( $query ) {
    if ( is_admin() || ! $query->is_main_query() ) {
        return;
    }

    $shelf = get_query_var( 'shelf' );
    if ( !isset( $shelf ) ) {
        return;
    }

    global $wpdb;
    $book_ids = $wpdb->get_col( $wpdb->prepare(
        "SELECT book_id FROM custom_shelf_table WHERE shelf_code = %s",
        $shelf
    ) );

    $query->set('post_type', 'pgn_book');
    $query->set('post__in', $book_ids);
});
add_action( 'pre_get_posts', 'pgn_filter_posts_by_shelf' );
```

This code:

1. Adds "shelf" to the list of whitelisted query vars.
2. Registers a rewrite rule transforming /shelf/a into index.php?shelf=a.
3. Modifies the main WP_Query object to include only posts with IDs fetched from the custom table.

Of course this example is completely superfluous. The fact that IDs of books on given shelves are stored in a custom table is the first thing that should raise your eyebrows. This functionality can be achieved perfectly with just a custom "Shelves" taxonomy.

How would you then make this URL structure work? Well, I'm pleased to say you wouldn't need to use the Rewrite API. You probably don't remember, but when you're registering a taxonomy

with register_taxonomy(), one of the $args options is "rewrite", which lets you set the slug of the taxonomy (it's the same for custom post types). Making this structure work would therefore be as simple as setting 'rewrite' => ['slug' => 'shelf'].

But let's assume you were an idiot and you wanted to flex your newly acquired Rewrite API knowledge in front of your friends. At the end of the day, the more complicated the code the better, am I right? I'm once again pleased to say that you wouldn't need to add a custom query variable. You could do all of that using only the default query vars that are already whitelisted and used in the main WP_Query. Here's the code:

```
// assuming "pgn_shelf" is the name of your taxonomy (WordPress automatically
registers it as a query var)
add_rewrite_rule( 'shelf/(.*)?', 'index.php?pgn_shelf=$matches[1]', 'top' );
```

This is also how my previous example of rewriting /post/post-name to index.php?post_type=post&name=post-name worked. Both 'post_type' and 'name' are default whitelisted query vars used in the core for creating the main query. As a matter of fact, it's very rare that you will ever need to work with custom query vars, which is why I had to resort to the overengineered custom table example.

How Rewrite Rules Are Stored

This part is important. Rewrite rules are not recomputed on every request. They are only calculated when you call flush_rewrite_rules() (which happens when you visit the Permalinks Settings page) and stored in the wp_options table with the "rewrite_rules" key. This option's value is just a huge associative array of regexes and their plain equivalents, e.g., array('shelf/(.*)?' => 'index.php?pgn_shelf=$matches[1]').

This array contains all rewrite rules used in $wp->parse_request() when trying to match the path. The fact that it is only regenerated on flush_rewrite_rules() means that you should always call add_rewrite_rule() on the "init" hook, which will be called on the permalinks settings page. Otherwise, your rules will not get registered when rewrite rules are flushed this way (calling add_rewrite_rule() does basically nothing if flush_rewrite_rules() isn't called during the same request, which is good).

But why are rewrite rules stored in the database? Why aren't they computed in code for every request? The reason is very simple - performance. There's more to computing rewrite rules than add_rewrite_rule(). The number of core rules is large and computing them all is expensive. The "rewrite_rules" option stored in the database is really big.

It would be extremely inefficient to recompute them every time, which is why you should absolutely never call flush_rewrite_rules() on every request (like on init). If your plugin adds custom rewrite rules, you should call flush_rewrite_rules() only on activation and deactivation

hooks. This way, the user won't have to manually flush them, and your plugin won't negatively affect the performance of their website.

Permastructs & Rewrite Tags

When I said that "there's more to computing rewrite rules than add_rewrite_rule()", I was mostly referring to permastructs. A permastruct (or a permalink structure) is just a set of rewrite rules generated for a common base path. This concept is a little complex so it'll be better if I lead with an example.

Think back to how we manually added a rewrite rule for our shelf taxonomy. That worked, but only for the first page… What if our taxonomy was paged, i.e., it allowed /page/2, /page/3, etc.? We didn't add a rewrite rule for pages, so they wouldn't work. That's exactly why you shouldn't do that manually, but more importantly, it shows why permastructs exist.

A permastruct is added using the **add_permastruct()** function. This function basically just registers a set of rewrite rules (you can specify which rules you want from a predefined list). It's the exact function WordPress core uses to add rewrite rules when you register a custom taxonomy or a custom post type.

The base of a permastruct is defined using **rewrite tags**. A rewrite tag is just a placeholder that represents a variable in the URL. Rewrite tags are always wrapped around percentage signs (%). As a matter of fact, you can see a few default rewrite tags when you visit the Permalinks Settings page, like %year%, %post_id%, %postname%, %category%, etc. A rewrite tag is registered with **add_rewrite_tag()**:

```
add_rewrite_tag( '%postname%', '([^/]+)', 'name=' );
```

The above is the exact call that registers the %postname% rewrite tag. What it means is that if you use %postname% in a permastruct, it will be replaced with "([^/]+)", and the captured value will be used as the "name" query var. That's what happens when you enable custom structure in permalinks settings and use /%postname%/. example.com/abc/ gets translated to index.php?name=abc.

Rewrite tags are meant to be used in permastructs. That's because add_permastruct() doesn't allow custom regex the same way add_rewrite_rule() does. A rewrite tag is just a single variable that is used to create a permastruct's base. A permastruct can use multiple tags, e.g., /%author%/%postname%/, which would become ([^/]+)/([^/]+) and would be mapped to index.php?author_name=$match[1]&name=$match[2].

Let's go back to our example of manually adding the shelf rewrite rule, but let's do it right this time - using add_permastruct() (it's still a bad idea btw):

```
function pgn_register_shelf_permastruct() {
    // Register the rewrite tag so WP knows how to turn %pgn_shelf% into a query var
    add_rewrite_tag( '%pgn_shelf%', '([^/]+)', 'pgn_shelf=' );

   add_permastruct(
        'pgn_shelf',           // unique name
        'shelf/%pgn_shelf%',   // structure template
        array(
            'with_front'    => false, // don't prepend with $wp_rewrite->front
        )
    );
}
add_action( 'init', 'pgn_register_shelf_permastruct' );
```

This will automatically add a family of rewrite rules, including:

- **/shelf/a/** -> index.php?pgn_shelf=a
- **/shelf/a/page/2/** -> index.php?pgn_shelf=a&paged=2
- **/shelf/a/feed/rss2**/ -> index.php?pgn_shelf=a&feed=rss2

Filesystem API

The Filesystem API is a family of WordPress functions and methods used to interact with the server's filesystem. It includes methods such as get_contents(), put_contents(), mkdir(), and more. The first question you might ask is - why? Why does WordPress have a proprietary filesystem API when functions like file_put_contents() exist in PHP? The short answer is - file ownership.

Let's assume that you uploaded your WordPress files with user deployer:deployer. All of your core files are owned by this user, but your web server runs as www-data:www-data. If you tried to create a file using PHP functions, the file would be owned by www-data:www-data. Suddenly, your WordPress files have mixed owners.

This creates a few problems. The deployer user might not be able to edit files owned by www-data (and vice versa). It can also introduce security questions in certain badly configured shared hosting environments, where multiple websites are run by the same web server, and when you don't want code run by your web server to be able to modify crucial files (like wp-config.php or .htaccess). The Filesystem API solves these problems by switching to FTP when file ownership mismatch is detected (more details soon).

When To Use The Filesystem API

Let me start by saying that the likelihood you will ever need to use this API is incredibly low, even if your plugin needs to create files. The Filesystem API was originally added in WordPress

2.6 to support WordPress's own automatic update feature. To this day, it is used whenever you update or install a plugin or theme.

When you're unsure if you should use this API, ask yourself this question - "Is the file I'm creating important enough to be version-controlled?". If the answer is no, it's likely that you can use built-in PHP functions directly. The essence of the question is that not every file needs careful ownership management.

Examples of use cases where the Filesystem API is **not** necessary:

- Creating files in a directory that is inherently supposed to contain "content", e.g., /wp-content/uploads.
- Creating disposable files, e.g., logs, cached pages, temporary files, exports, backups, etc.

Examples of use cases where the Filesystem API **is** necessary:

- Creating/modifying core WordPress files, e.g., in /wp-admin/ or /wp-includes/ (don't).
- Creating/modifying files that are part of the "application code", e.g., plugin files with a proprietary plugin update system.
- Creating/modifying important configuration files, e.g., wp-config.php, .htaccess, etc.

The underlying distinction is between critical application files and runtime data/user content. Application files have different security expectations. It is usually not desirable for your web server to be able to tinker with core WordPress files (even though that's the default if you use the one-click WordPress installer). Similarly, mixed owners within those files might prove problematic if a deployment script is used and one of the version-controlled files has a different owner on the server. In such a case, the script would fail.

In contrast, files like debug.log or temporary full-page caches are supposed to be created and modified by the web server. They do not contain critical site code and are usually not included in version control. Having those files not be writable by the web server would mean that for every write operation, WordPress would need to connect to the server through FTP as the user that owns these files. That's why WordPress itself doesn't use the Filesystem API for uploading files to /wp-content/uploads.

How It Works

You already know the most important thing you can - why it exists and when to use it. Trust me, I wasn't able to find a single good explanation of that anywhere. It's time you see how it works.

The core of the API is located in the /wp-admin/includes/file.php file. This file is inside the wp-admin directory because the API is supposed to only be used in the admin panel. It is not loaded on the frontend by default.

Four connection types are supported and chosen in the following order of preference:

1. **Direct** (using built-in PHP functions).
2. **SSH2**.
3. **FTPS**.
4. **FTP**.

The beauty of this API is that it provides a unified interface for all these different connection types. The API follows the Singleton pattern, where the global $wp_filesystem is an initialized object for the selected connection type. All connection type classes extend the WP_Filesystem_Base class. This class includes methods like:

- **chmod()**
- **chown()**
- **copy()**
- **get_contents()**
- **mkdir()**
- **put_contents()**
- and many more…

To use this API, you can call methods like: $wp_filesystem->put_contents($file, $contents). That being said, the $wp_filesystem object is not initialized on every admin request. It is null by default. To initialize it, you have to manually call request_filesystem_credentials(). This function is the most confusing, yet the most important part of the Filesystem API.

I'm going to start with a short summary of how it works, and then we'll get to the step-by-step initialization process. request_filesystem_credentials() returns an array of credentials you pass to WP_Filesystem() to initialize $wp_filesystem. It checks if either direct or SSH2 can be used. If they can't, it tries to find FTP credentials defined in wp-config.php. If there are none, it outputs an HTML form asking the user to input the credentials.

Below is a full step by step process of initializing $wp_filesystem (the most complicated part of this API). I will show you a concrete code example shortly after.

1. The user triggers some action that requires the Filesystem API (e.g., clicks the "update" button).
2. The code hooked to run on this action (e.g, on this POST request) starts off by calling request_filesystem_credentials() and storing its return value in $creds.
3. request_filesystem_credentials() does the following:
    a. Try to find the credentials in $_POST.
    b. Check if the direct method can be used by creating a temporary file using fopen() and comparing its owner to the owner of /wp-admin/includes/file.php.
    c. If direct can't be used, check if constants like FTP_HOST, FTP_USER, and FTP_PASS are defined in wp-config.php.
    d. If constants aren't defined, render an HTML form asking the user to choose a connection type (SSH/SFTP/FTP if appropriate PHP extensions are loaded on the server) and input credentials.

    e. If the form is rendered, return false. If credentials are obtained, return an array containing them.
4. If $creds is false, it means that the form has been rendered. We call wp_die().
5. If $creds is not false, we pass it as an argument to WP_Filesystem(). If this function returns true, it means $wp_filesystem has been successfully initialized. If it returns false, it means the credentials were incorrect.

There are a few interesting things here. First of all, you can see that $wp_filesystem should usually be initialized on interactive user action, e.g., submitting a form, clicking a button, etc. That's because there is a possibility that the user will have to fill out a form.

The request_filesystem_credentials() function starts by looking for the credentials in $_POST. It is meant to find them after the user has submitted the form it previously rendered. It will only ever find them on the "second execution" of request_filesystem_credentials(). This means that you need to ensure the original code that you ran after the user has clicked the update button, also runs after they submit the credentials form.

Thankfully, request_filesystem_credentials() allows you to specify the URL the form will be POSTed to, along with additional hidden input fields, which you can use to preserve the data POSTed by the user in their original request (when they clicked "update"). Here's a simple visualization if you don't understand:

1. The user clicks "update". A POST request is sent to the server with "plugin_id" = 1. Your code only executes if "plugin_id" is present in $_POST and if the POST is sent to /wp-admin/update, so it's going to execute.
2. Your code calls request_filesystem_credentials(). It specifies /wp-admin/update as the path the form will POST to (its action attribute). It also specifies "plugin_id" = 1 in the extra fields parameter, which will render that as a hidden input field.
3. The user fills out the credentials form and submits. It POSTs to /wp-admin/update, and the "plugin_id" field exists, so your code runs again. request_filesystem_credentials() is called (again), but this time, it finds the credentials in $_POST, so it doesn't have to render the form.
4. $wp_filesystem is initialized and you can use it in the rest of your code.

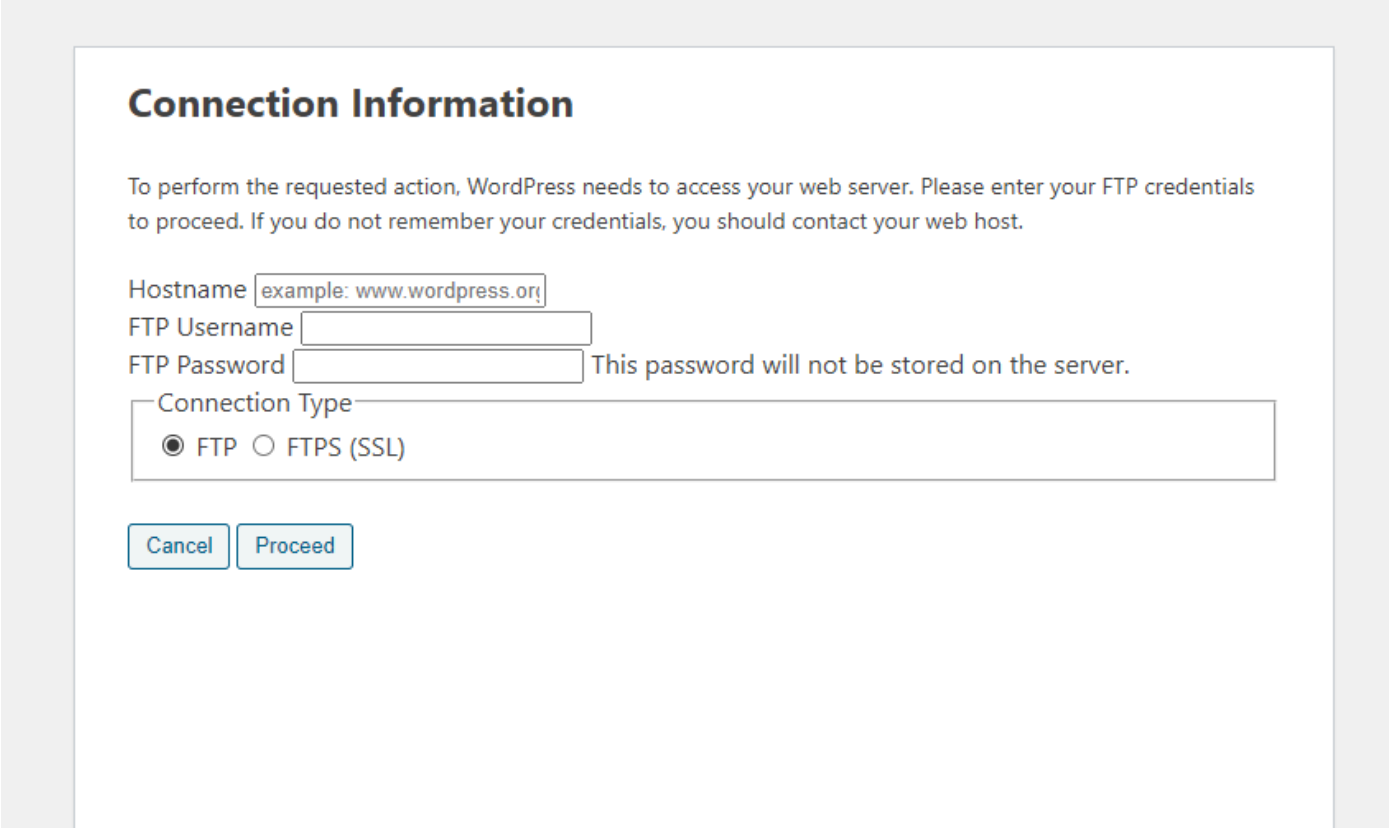
Code Example

Here's a code example roughly following what I just described (minus the URL path check and POSTing to /wp-admin/):

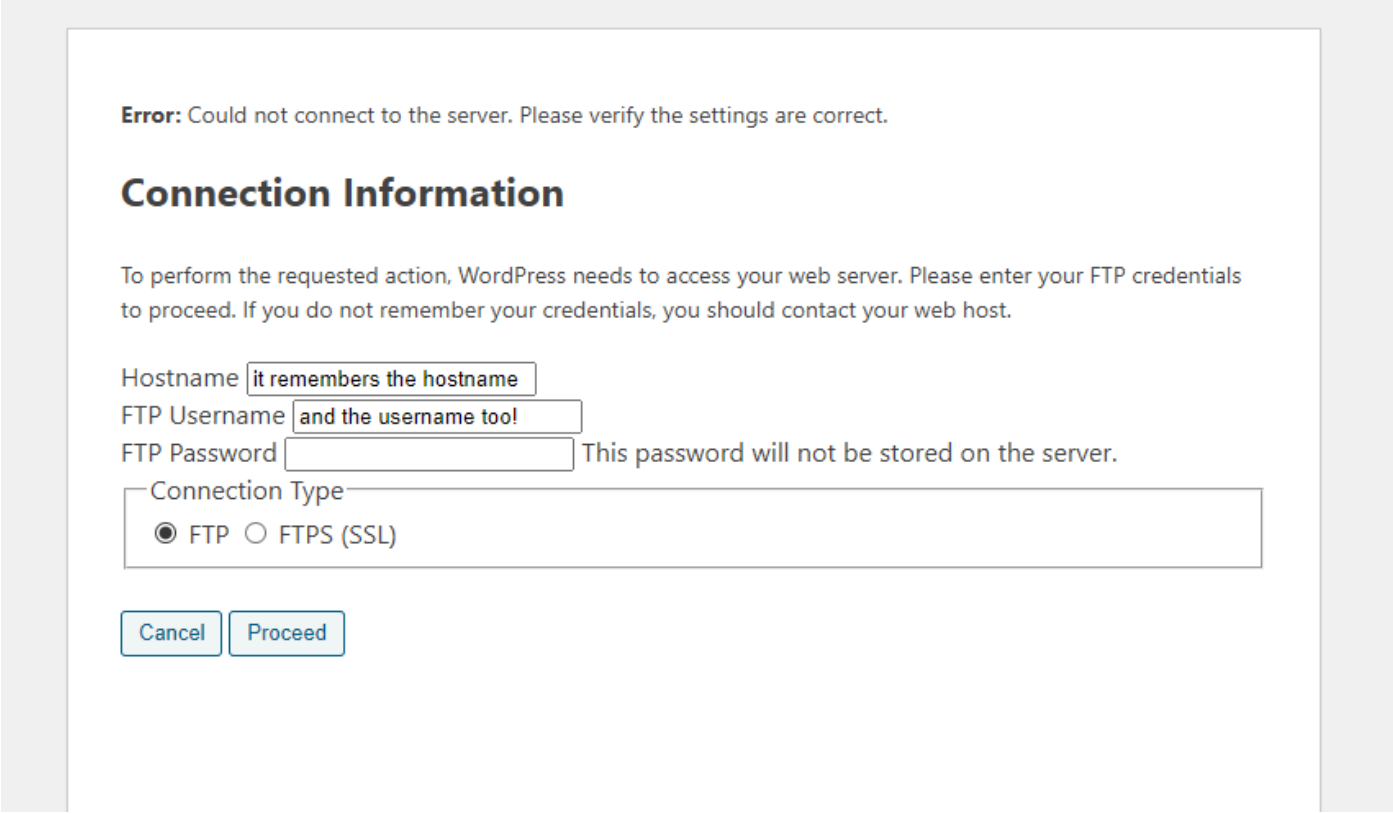
```
function fsapi() {
    if ( ! isset( $_POST['pgn_plugin_id'] ) ) {
        return;
    }
```

```
    global $wp_filesystem;
    $creds = request_filesystem_credentials(
        admin_url(),
        '',
        false,
        '',
        array( 'pgn_plugin_id' => $_POST['pgn_plugin_id'] ) // This will become a
hidden form field.
    );

    // Direct couldn't be used and credentials constants weren't found.
    if ( ! $creds ) {
        wp_die();
    }

    // Credentials were incorrect so we're setting the "error" argument to true.
    if ( ! WP_Filesystem( $creds ) ) {
        request_filesystem_credentials(
            admin_url(),
            '',
            true,
            '',
            array( 'pgn_plugin_id' => $_POST['pgn_plugin_id'] ) // Still preserving
original data.
        );
        wp_die();
    }

    // $wp_filesystem is initialized, we can proceed.
    $wp_filesystem->put_contents( plugin_dir_path( __FILE__ ) . 'test.txt', 'abc',
FS_CHMOD_FILE );
}
add_action( 'admin_init', 'fsapi' );
```

You can see that we're hooking this function to admin_init, but we're only actually running the code if "pgn_plugin_id" is present in the $_POST. The return value of the first request_filesystem_credentials() is captured to $creds and checked. It's going to return false at first because my /wp-admin/includes/file.php file has a different owner than www-data and I don't have constants configured. Here's what the user (admin) will see after submitting the form with "pgn_plugin_id":



This form will be the only thing rendered on the entire page (because we called wp_die()). You can see that there are only two connection types - FTP and FTPS. There's no SSH because I don't have the ssh2 PHP extension enabled on my server. Look back up at the code. When $creds is not false but WP_Filesystem() fails, it means that the credentials were wrong, so we request them again, this time passing the error flag. Here's what the form will look like after submitting bad credentials:



You can see "Error: Could not connect to the server. Please verify the settings are correct." at the top of the form. If $wp_filesystem gets initialized successfully, we write "abc" to the file test.txt inside the current plugin's directory.

Using The Filesystem API In Non-Interactive Environments

Up to now, we've only talked about using the Filesystem API in an environment where it's the user that prompts the action. They submit a form or click a button. This is an interactive environment, because we can render the form and expect the user to be able to fill it out. But what if that's not the case? What if, for example, we had to use the Filesystem API in a cron job?

In such a case, you would not be able to depend on the user input. The only way you can use this API like that is by forcing the user to define their credentials as constants in wp-config.php. You would also have to suppress the rendering of the form (like with output buffering) and throw a definitive error if the user has failed to define the constants. Just bear in mind that it's more complicated and non-standard, but not impossible.

WP-Cron

WP-Cron is WordPress's userland implementation of cron. I'm assuming you know what cron is - it's a system for executing scripts at a configured time. It is used to execute scheduled recurring tasks. Similarly to how WordPress nonces aren't real nonces and salts aren't real salts, WP-Cron isn't a real cron. It's an imitation of a cron limited by the nature of HTTP.

The limitation I'm referring to is the fact that a PHP process is only instantiated when someone sends a request to the server. Otherwise, WordPress code doesn't run. This has a few interesting consequences. First of all, WP-Cron can't promise execution at any exact time. What if you set your cron to run at 12 am on every Friday, but no one visits your website at that time? No PHP process would be created, WordPress code wouldn't run, and your task wouldn't be executed.

The way WP-Cron interval works is super confusing and you need a little more context to understand it. Bear with me and I promise you will get it by the end of this chapter.

Some examples of when you would need to use a cron:

- Automatic background security scan.
- Scheduled backup.
- Periodic synchronization between your database and an external API.
- Automatic periodic clearing of temporary files and cache.

How WP-Cron Works

WP-Cron utilizes the hook system. A cron job in WordPress is not a callback or a file. It's an action. You register this action using wp_schedule_event() and passing its hook name and the interval. WordPress then checks if this interval has passed on every request and, if it has, it calls do_action(). The real "cron job" (the code that runs) is just the callback (or callbacks) you hook into this action.

Scheduling Events

The above was an oversimplification. Let's get into the details. wp_schedule_event() schedules a recurring event. It accepts the initial timestamp (when the action should first be called), a recurrence string, the hook name, and an optional array of arguments to be passed to the callbacks hooked to the action. Here's an example:

```
wp_schedule_event( time(), 'hourly', 'pgn_synchronize_database' );
```

This will schedule a task to call the "pgn_synchronize_database" action every hour starting right now (from the very next request). You would then hook a callback to this action with

add_action('pgn_synchronize_database', 'pgn_callback'). The function pgn_callback() will be executed roughly every hour. Very important - a recurring event should be scheduled only once (e.g., on plugin activation). More on that later.

You might be surprised to see "hourly" instead of HOUR_IN_SECONDS or 3600. WP-Cron intervals are dependent on registered intervals defined as strings. Default intervals are "hourly", "twicedaily", "daily", and "weekly". You can define custom intervals by hooking into the "cron_schedules" filter, like so:

```
function pgn_add_fifteen_minute_interval( $schedules ) {
    $schedules['fifteen_minutes'] = [
        'interval' => 900, // 15 minutes in seconds
        'display'  => __( 'Every Fifteen Minutes' ),
    ];
    return $schedules;
}
add_filter( 'cron_schedules', 'pgn_add_fifteen_minute_interval' );
```

This will allow you to use the "fifteen_minutes" interval. You can see why this design choice was made in the above code example. Every interval has a "display" attribute. This attribute was originally created so that intervals can be displayed to admins in a human readable form. It's better UX to choose between "Hourly" and "Daily" than it is to choose between "3600" and "86400".

PS: There's also wp_schedule_single_event(), which schedules a one-off, non-recurring event.

How Events Are Stored

Events are not defined purely in code. They are stored in the database, and they have to be because they are inherently stateful. They are just a serialized array of all scheduled actions. This array is stored as an option (in wp_options) with the name "cron". Here's an example of this array:

```
array(
    '1759001996' => array(
        'pgn_synchronize_database' => array(
            '40cd750bba9870f18aada2478b24840a' => array(
                'schedule' => 'hourly',
                'args'  => array(),
                'interval' => 3600
            )
            // Other events at the same timestamp with the same hook, but with
different args
```

```
        )
        // Other hooks scheduled to run at the same timestamp
    ),
    // Other events scheduled at different timestamps...
);
```

Okay, there's a lot to untangle here. The first top-level key is the UNIX timestamp of when the event should next be run. WordPress checks these timestamps on every request. If it just so happens that it's smaller than the current timestamp (i.e., it's in the past), WordPress executes the event and updates the timestamp to when the event should next be executed.

The value of this timestamp is another associative array. Its keys are the hook names. Notice that if you scheduled two events at the same time but with different hooks, they would both be here. The value of that hook name is yet another array of arrays (it's a lot of nesting, I know).

The key (40cd75…) is the MD5 hash of the "args" attribute. That's the optional array of arguments you define when calling wp_schedule_event() that is passed to the action callbacks. It means that you can schedule the same event, at the same time, with the same interval, and only vary the args, and it will be treated and executed like a separate event. In our case, the MD5 hash is 40cd75… because that's the hash of an empty array.

The values in the array under this MD5 hash are the details about this exact event. You can see that there are both the string "schedule" and the integer "interval". The integer value is used as a fallback if the string isn't defined with the "cron_schedules" filter (i.e., when it was there when the event was first scheduled but later got removed for some reason).

I said earlier that I would go back to why you should only schedule an event once. You should now be able to understand why. If you called wp_schedule_event() on every request (e.g., hooked to 'init') with time() as your initial timestamp, every request would schedule a new event.

If you received one request every second for 10 seconds, that would schedule 10 new events. They would all call the same hook with the same arguments - the only difference between them would be the timestamp (the difference would be 1). If it was an hourly event, the same event would be called 10 times every hour (and ten seconds). And that's assuming you remove the offending code after 10 seconds. If you don't, new events will keep getting scheduled on every request, eventually leading to your tasks being called every second.

The bottom line is - wp_schedule_event() adds the event to the option in the database. Once that happens, this event and its state (when it's next going to be executed) lives and is updated entirely in that database. That's why you should usually schedule events on plugin activation or user action and always check if this event isn't already scheduled with wp_next_scheduled(), like:

```
// Only schedule if the event doesn't yet exist
if ( ! wp_next_scheduled( 'pgn_synchronize_database' ) ) {
    wp_schedule_event( time(), 'hourly', 'pgn_synchronize_database' );
}
```

WP-Cron Lifecycle

Now that you know what cron events are and how they work, it's time to tie this entire system together by walking through it step by step.

1. An Event Is Scheduled

The administrator installs and activates a new plugin. This plugin calls wp_schedule_event() on its activation hook. The initial timestamp passed to wp_schedule_event() is time(), which is just the current UNIX timestamp. That's the timestamp the event will be stored in the database with.

2. A User Visits The Website

Someone visits the website (it doesn't matter who). WordPress executes its typical boot process and calls the 'init' action when it's finished. The wp_cron() function, which is hooked to this action, executes.

3. WordPress Checks If There Are Any Events Scheduled To Run

wp_cron() retrieves the "cron" option containing the array of all scheduled events. This array is sorted by timestamps (the lowest timestamps are first, i.e., the ones most in the past). wp_cron() iterates over this array and if it finds any timestamp that is in the past, it calls spawn_cron() and stops iteration. Otherwise, if the first timestamp in this array is in the future, it means that no events are scheduled to run.

4. WordPress Checks & Acquires A Lock

WordPress checks if there is an existing valid lock. If there is, WP-Cron execution is aborted and the rest of the normal request lifecycle happens. If there isn't, it acquires one and proceeds.

A lock is just a timestamp stored as a temporary value in the database (a transient). It's used to prevent race conditions. If WP-Cron didn't have a locking mechanism, a request made one second after another request would pose the risk of running the same event twice (beginning it before the first one updated the timestamp).

A lock is valid for a given amount of time defined by the WP_CRON_LOCK_TIMEOUT constant. It's 60 seconds by default. This means that if a single cron job runs for more than 60 seconds, the lock will be deemed invalid and a new cron job may be started by a new request (with the old lock being overwritten by the new one).

5. WordPress Sends A Request To wp-cron.php

spawn_cron() utilizes the HTTP API to send a request to /wp-cron.php?doing_wp_cron=$timestamp. It uses the wp_remote_post() function. There are two extremely important args passed to this function:

- 'timeout' => 0.01
- 'blocking' => false

This means that the request is asynchronous. It doesn't wait for a response. The lifecycle of WP-Cron for this request ends. WordPress handles all further processing as usual and returns a rendered page to the user.

The next step in WP-Cron's lifecycle is now in wp-cron.php, which is a completely separate process running on the server. This is why WP-Cron doesn't have a noticeable impact on the page's load time for users that trigger it.

6. wp-cron.php Executes Scheduled Events

wp-cron.php starts off by checking if its lock (the doing_wp_cron timestamp query parameter) is the same as the lock in the database. If it is, it retrieves and iterates over all events in the "cron" option until it reaches one in the future (remember - they are ordered).

For every event iterated over that should be run:

1. it reschedules the event with wp_reschedule_event();
2. it unschedules the current event (with the timestamp in the past) with wp_unschedule_event();
3. it calls do_action_ref_array($hook, $event['args']), executing the callbacks hooked to this action.

wp_reschedule_event() creates a new event with all the same details but a different timestamp. It uses wp_schedule_event() internally. The timestamp is calculated so that it aligns with the initial timestamp. If you originally scheduled this event to run at exactly 6 pm, and it runs at 6:30 pm because there was no traffic, wp_reschedule_event() will schedule the next execution for 7 pm, not 7:30 pm.

WordPress cares more about preserving the exact schedule time than it does about preserving the interval between executions. This means that if your event is scheduled to run hourly at a full hour (i.e., if your initial timestamp was exactly 3:00 pm), and your first request between 3 and 4 pm comes at 3:58 pm, the event will be rescheduled to 4:00 pm - exactly two minutes after it was run.

This characteristic means that you can't count on your interval at all. It is neither the maximum nor the minimum interval between executions. An event scheduled to run every 24 hours can not run for 47 hours, and then run twice in one hour. This is obviously an extreme example with literally 0 requests to the site in 23 hours, but you get the point.

7. wp-cron.php Checks/Deletes The Lock

A single action callback can take a long time. Let's say you're taking a full site backup and sending it to a remote storage. This may take longer than 60 seconds (which is the default lock timeout). After calling each event's action, it checks if the lock in the database is still equal to the one it uses. If it's not - it means that another cron request has taken over and is already working on executing the rest of the scheduled events. Execution is aborted.

If wp-cron.php manages to finish executing all scheduled events, the lock is deleted from the database and die() is called. Because there is no lock anymore, another cron job will be started on the next user request if there's any event that is scheduled to run (with the timestamp in the past - somewhere between when the previous job started its execution and now).

Unscheduling Events

An event won't unschedule itself when your plugin is deleted (duh!). Please don't leave junk behind yourself. Always call wp_unschedule_event() for all your events on the deactivation hook.

Making WP-Cron More Reliable

WP-Cron is pretty unreliable, but given the constraints of the environment, it's the best you can natively get. If your site has very few visitors, you will find that the difference between when events are scheduled to run and when they actually are run is large.

It can be problematic even if your site does have a lot of visitors. If your website is static, like a blog or a company brochure, you will almost certainly use full-page caching (we will cover that soon). This will circumvent the execution of the WordPress lifecycle, including the init hook. You can have thousands of visitors every hour, and yet, your cron will never run.

There is a common pattern used to make WP-Cron more reliable. You can create a cron job on your server's system that triggers execution of WP-Cron. This can be just a bash/powershell command sending a request to /wp-cron.php (without the doing_wp_cron parameter - WordPress will acquire a new lock).

You can schedule this job to run every minute on your server, and because your OS's scheduler is reliable and trustworthy, you can be sure that WP-Cron will be checked every minute - even if there's no traffic. But even if you do that, WordPress will still check and attempt to execute WP-Cron on every request. To disable that (and save on some processing), you can put this in wp-config.php:

```
define( 'DISABLE_WP_CRON', true );
```

This will disable step 3 in the lifecycle, meaning that WordPress will not check for scheduled events on every request. That's fine because your server-side script is hitting /wp-cron.php every minute anyway. I haven't run any benchmarks, but if you opt to disable WP-Cron, you might save some memory and processing time by disabling autoload of the 'cron' option in the database. This will prevent it from being prefetched and then not utilized, and trust me - this option can get quite large.

Alternative Solutions

WP-Cron might not always be the best choice. There are some alternative cron systems like Cron-Control, Cavalcade, and Action Scheduler (which you can bundle with your plugin) that are more sophisticated and better optimized for large websites. These usually utilize separate database tables, allow for parallel execution, retries, and more. They are outside the scope of this guide - go check them out if your use case calls for a better cron (or job queue) system.

ALTERNATE_WP_CRON

There's an alternative mode to the built-in WP-Cron. You can activate it by defining the ALTERNATE_WP_CRON constant as true (i.e., in wp-config.php). This mode works by redirecting the user who triggered the cron to the same URL with ?doing_wp_cron query parameter appended. A request to any URL with this query parameter will trigger a cron check.

Note that this absolutely will have a negative effect on the load time for this user, as the server needs to send a redirect and they have to request the new page again. This mode should only be used as a fallback if loopback requests (i.e., your server calling its own /wp-cron.php) are disabled in your hosting environment. I'm including it more as a fun fact so that you know what's going on when you see a WordPress website redirecting you to URLs with ?doing_wp_cron.

Caching

At last, caching. The source of headaches and broken dreams if not understood, and one of the biggest levers you can pull to make a website faster. It can be the difference between an unusable mess and a heavenly smooth experience. I don't believe you can call yourself a WordPress developer (or a web developer for that matter) if you don't have a deep understanding of caching.

This chapter will be a little different than other chapters. The underlying rule I established in the introduction was that I tend to not discuss things outside the scope of a "WordPress deep dive". However, caching is such an important and wide topic that I'm going to break this rule. I will cover all of the layers of caching most commonly seen in professional WordPress websites. That being said, I will not dive deeply into non-WordPress sections. I will only mention them, and the rest is on you.

But let's take a step back. What even is caching? Caching is the process of storing copies of data in a temporary, high(er)-speed location (a cache). Its purpose, at the end of the day, is to make your website faster. The most direct positive impact is achieved by caching expensive and long operations, such as database queries, heavy PHP computation, blocking (synchronous) API requests, etc.

Web caching is layered. It starts at the level of PHP execution and ends on your users' machines. The following sections attempt to walk through these layers from the very bottom to the very top. By the end of this chapter, you should have a decent understanding of all of these layers, and an excellent understanding of the layers specific to WordPress.

1. PHP OPcache

"OPcache improves PHP performance by storing precompiled script bytecode in shared memory, thereby removing the need for PHP to load and parse scripts on each request." ~ official PHP documentation. This is way outside of the scope of this guide, but as promised, I'm mentioning it.

What it means is that your PHP files won't have to be parsed and compiled on every request. They are compiled just once and their bytecode is stored in memory. In reality, you'll usually not have to worry about it unless you're running WordPress on a VPS. Even then, OPcache seems to be enabled by default in php.ini.

2. WordPress Object Cache

One important global variable in WordPress is $wp_object_cache. It's an object of class WP_Object_Cache. This object's job is basically just being an associative array of cached values. Imagine you had some expensive query or heavy computation to perform multiple times during the page load. Instead of having to do them over and over again, you just store their results in this cache and reuse it later.

You should almost never interact with this object directly. Instead, use one of the many helper functions. The most important ones are:

- **wp_cache_get()**
- **wp_cache_set()**
- **wp_cache_add()**
- **wp_cache_replace()**
- **wp_cache_delete()**
- **wp_cache_flush()**

It's important to understand that none of this is automatic. You have to consciously decide what data to cache and then actually use the appropriate functions. Let's say you had some expensive database query to make and wanted to cache it in Object Cache. Here's what that would look like:

```php
// Start by trying to read the value from the cache.
$result = wp_cache_get( 'expensive_query', 'post-reading-time', false, $found );

// If the value isn't there, do the expensive operation and cache it.
if ( ! $found ) {
    $result = $wpdb->get_col( $query );
    wp_cache_set( 'expensive_query', $result, 'post-reading-time' );
}
// Use $result as usual.
```

A few things to cover here. "expensive_query" is the cache key. It's basically the key in the associative array. "post-reading-time" is a group. A group is like a namespace for cache keys. Its purpose is to prevent name collisions and group different parts of the Object Cache together. It's the reason why we didn't have to prefix "expensive_query" with "pgn_". But why did I make the group "post-reading-time" and not "pgn"? Well, it's a good practice to name your group like your plugin slug. This is similar to how the text domain in i18n functions should also be the slug.

The fourth argument to wp_cache_get() is particularly interesting. It's a variable passed by reference. Its value will be true if the cache key was found, and false if it wasn't. We later use it to decide if we should perform the expensive operation. In the real world, you'll often see people use only the key and the group, and then check if $result is false to determine if the value was found. This is incorrect (usually). The reason is that the cached value can be falsy (or even just false).

If you do "if (! $result)", how do you know if the value wasn't found, or if it was found but was 0? You're going to do the heavy processing and get 0 again, even though you already had it in the cache (even strict comparison won't protect you if the cached value can be false). On the other hand, $found will always be either true or false - depending on if the value was found. As a rule of thumb - always use the fourth parameter. Even if you have 100% certainty that the value can never be falsy, it's just a more robust practice.

There is one huge problem with the default Object Cache - it's not persistent. Because it's just an associative array in memory, it gets destroyed after PHP is done processing the request. All of its data has to be regenerated on every request. This means that if you were to do the expensive operation only once, you'd see absolutely no performance benefit from using cache. Thankfully, this is a solved problem.

Persistent Object Cache

Ever heard of Redis or Memcached? These are basically external in-memory associative arrays/databases. You run them as separate processes on your server and connect to them through the network (just like you do with your database; IP 127.0.0.1 if the Redis/Memcached server is running on the same machine as your web server). Because of that, they aren't

dependent on your PHP processes running. This means they don't get wiped at the end of every request, i.e., they are persistent.

WordPress provides an easy way of replacing the built-in cache system with a persistent one. Before loading this cache API and instantiating $wp_object_cache, WordPress looks for a file called object-cache.php in /wp-contents/. If this "drop-in" file is present, it is loaded and used instead.

That's the foundation of many plugins, like Redis Object Cache, Object Cache Pro, SQLite Object Cache, and more. All of these plugins create this file when activated. The file contains their own implementation of the WP_Object_Cache class and the procedural helper functions. The most important part is - the signatures of the functions stay the same, so you don't have to modify your code after installing a plugin like that. You just activate it, configure it, and boom - your site has persistent Object Cache.

Keep in mind that for this to work, you have to have the Redis server (or whatever you're using) running somewhere in your environment. It's a program external to your website that the plugin has to connect to to read from and write to. Keep that in mind when choosing your hosting provider - most high quality ones will have it pre-configured for you.

This is great because it provides persistence and sharing between requests. Remember the expensive query you had to run just once for each request? You won't have to run it at all anymore as long as it's present in the cache. This really is life-changing, but there is a catch - you have to pay more attention to how you use the cache.

See, persistent cache introduces some problems its more primitive alternative doesn't have to worry about. First and foremost - you have to be careful about what data you cache. Imagine that you have a WooCommerce store and your code runs something like this:

```
wp_cache_set( 'user_billing', $user_data, 'myplugin' );
```

You just cached the personal billing information of the first user who happened to enter the website when the cache was empty. You will then likely keep using this information with wp_cache_get() and every other user in your store will see this billing info. That's a data privacy violation of the highest order. You didn't have to worry about that with non-persistent cache because it was specific to the request, but now that it's shared, you need to be very careful about what you cache.

When caching data like that, you have to make the key dynamic. Instead of "user_billing", make it "user_billing_$userid". You can also make the group dynamic, so instead of "my-plugin", make it "my-plugin-$userid". If you do that, you will also be able to flush all cache for the user with one call to wp_cache_flush_group() (if the Object Cache implementation you're using supports it). Whatever you do, just be consistent (for your own sanity).

Another problem with persistent Object Cache is data staleness. The wp_cache_set() function has an optional fourth argument - $expire (Time To Live - TTL). It expects a number of seconds signifying the time after which the entry should be deleted. The default value is 0, meaning indefinite (the data will attempt to stay in the cache forever).

So, what TTL should you use? This question is as if you came to my place and I asked you where the glasses are - how the hell are you supposed to know? You should know what TTL your data needs based on what data it is. Is it an API response that refreshes every 24 hours? How about 24 hours? Or calculate it dynamically to be invalidated at a given time. Is it something that needs to be relatively fresh? Use your judgment - maybe 5 minutes?

That being said, you should usually not set $expire at all. A much better strategy for cache invalidation is event-based. Just think about it - how long is the cache of user's billing info valid? Well, it may be 20 seconds, or it may be a year. The problem is that we're thinking of it in terms of the wrong units - time. In reality, it's valid until it's changed. Always try to invalidate the cache when the data is modified. Hook into the action fired when the data is changed and call wp_cache_replace() (or wp_cache_flush_group() if you need to flush a whole group of entries, e.g., all queries for posts when a new post is added).

How WordPress Core Uses Object Cache

The core WordPress code is full of calls to wp_cache_*(). This is why installing a persistent Object Cache plugin speeds up even vanilla WordPress installs. WordPress usually handles cache invalidation pretty well (usually in event-based fashion), but if you're using persistent cache and are experiencing some weird issues, spending a few minutes reading the core to see how it works probably won't hurt.

Some of the data WordPress caches includes:

- **Options** - get_option() tries to get the data from cache before querying the database.
- **Posts** - functions like get_post() try to get the post from the cache before touching the database. This cache is intelligently invalidated when the post is updated.
- **Metadata** - get_post_meta() and other metadata functions read from and write to cache.
- **Terms and Taxonomies**
- **Comments**
- and more…

As far as I know, full results of WP_Query calls are not cached by default. If you want your queries cached, make sure to handle that yourself (like we did with 'expensive_query') in the example above.

3. Transients API

The WordPress Transients API is very similar to Object Cache. As a matter of fact, it uses wp_cache_*() functions if persistent Object Cache is configured. The difference is that if it isn't configured (which is the default WordPress configuration), the value is stored in the database. This allows for persistent caching without a Redis/Memcached server.

There are 3 main functions provided by this API:

- **get_transient()**
- **set_transient()**
- **delete_transient()**

Every transient has an expiration time (TTL). You set that TTL when calling set_transient(). Alternatively, a transient can be deleted on an event with delete_transient(), just like you can delete an Object Cache entry with wp_cache_delete(). Here's an example of using the Transients API to cache an expensive remote API call:

```
// Start by trying to read the value from the cache.
$response = get_transient( 'pgn_response' );

// If the value isn't there, do the expensive operation and cache it.
if ( ! $response ) {
    $response = wp_remote_get( $url );
    // You should probably do some error handling here.
    set_transient( 'pgn_response', $response,  DAY_IN_SECONDS );
}
// Use $response as usual.
```

This is literally just the example from the Object Cache section with a few tweaks. You can see that transients don't utilize a group. We have to prefix our key with "pgn_". After this code runs (and assuming no valid 'pgn_response' transient exists), the 'pgn_response' transient will be added and its TTL will be 86400 seconds.

If a persistent Object Cache plugin is installed and configured, this entry will be stored in the Object Cache (Redis/Memcached). If it's not, the data (HTTP response) along with the expiration time will be stored in the wp_options table in the database. That's right, transients are stored in wp_options. As a matter of fact, the Transients API functions use the Options API internally, with calls to functions like get_option(), add_option(), and delete_option() all over the place.

Transients In The Database

Every set_transient() (without persistent Object Cache enabled) adds two options:

- **_transient_$key**

- **_transient_timeout_$key**

For our pgn_response, it would be _transient_pgn_response and _transient_timeout_pgn_response. The first one is obviously just the value you're storing. In our case, it'd be the serialized array/WP_Error object returned by wp_remote_get(). The timeout is calculated in set_transient() as time() + $timeout. It's just a UNIX timestamp of when the entry should be considered expired.

Expired transients are deleted from the database when you try to fetch them with get_transient(). This used to be the main method for purging transients. However, you can probably see why it's flawed. What if the code calling get_transient() never runs? Like if you delete a plugin after it has set a transient. In such a case, the transient would just stay there forever, littering your wp_options table.

Thankfully, this has changed in WordPress 4.9. This update introduced the delete_expired_transients() function and a WP-Cron event with the same name. This event runs once a day and calls this function. As a side note - make sure you don't have persistent Object Cache configured if you're trying to call delete_expired_transients(). I was seriously puzzled as to why this cron event wasn't working, only to realize that it doesn't purge the transients from the database if Object Cache is in use (even if they are expired).

Transients vs Object Cache

The Transients API might look like a better Object Cache. Why use wp_cache_*() functions if you can just set and get transients? It will use persistent Object Cache anyway, and will fall back to the database if it's not present. You can even use delete_transient() to flush an entry on an event. It seems kind of awesome, doesn't it? Well, it's complicated.

The first and most important rule you have to keep in mind when writing a public plugin or theme is that you should assume persistent Object Cache is not present. Most WordPress websites that install your plugin won't have Redis running on the backend. Your transients will end up in the database. This underlying assumption is the most important reason why transients aren't always the best choice.

Imagine what would happen if WordPress core used set_transient() to cache WP_Post objects for get_post(). On a site with 10,000 posts, you'd have an additional 20,000 options (the value and the timeout). This fact alone would make operations on wp_options slower, which would have a negative performance impact not only on transients, but on all Options API calls on your site.

Another obvious question is "why?". Why store a cache of the data in the database if the source of truth for this data lives in the same database (just in another table)? It may even take longer to fetch the transient than the data itself because transients require querying for two rows (value and timeout). Using transients for all post objects would actually increase the number of database queries, not reduce it.

That's exactly why WordPress doesn't use transients for all these objects like posts, options, terms, or comments. They would bloat up the database and most likely degrade performance. For such data, using the Object Cache makes perfect sense. A read from memory is almost always going to be faster than a query to the database, and even if no persistent cache is used - it doesn't hurt.

When considering whether you should use the Transients API or Object Cache, ask yourself if reading the data from wp_options will be noticeably faster than just computing it, and if it's not going to bloat the database. If you're caching something that's already in the database - using transients is silly. It's also irresponsible for you to use the Transients API for data that is dependent on an unbounded number of entities, like one entry for every user or post. A site with 100k users will have to deal with 100k transients.

On the other hand, caching a single HTTP response from a remote API is a perfect use case for a transient. It will be beneficial even if persistent Object Cache is not used and the transient lands in the database. At the end of the day - the performance question is really dependent on context. Is it going to be advantageous to cache the results of a complex mathematical computation in the database, or will the SQL query take longer than just computing it on every request? Run some benchmarks, analyze the trade-offs, and make a conscious decision.

There's also a question of semantical meaning. The Transients API was designed around the concept of entries expiring after some time. This is not the assumption for Object Cache, which is often used to store entries indefinitely with invalidation on events. You should keep that in mind when choosing the right caching mechanism.

That being said, transients can be used for data stored indefinitely. If you set TTL to 0, the _transient_timeout_$key option will not be created. WordPress core uses this technique to cache the results of site health check and only replace it the next time the health check script runs. But there's a catch - transients with TTL 0 are autoloaded. Why? I have no idea. Someone just decided to do it this way, and now all your transients without expiration will be loaded on every request, even if they aren't used.

If your situation really calls for an infinite transient (particularly if its multiple transients), it might be a better idea to set its TTL to some obscenely large value, like 10 * YEAR_IN_SECONDS. This way it won't be autoloaded. That being said, the Transient API is usually meant to be used with values that expire after a set amount of time, unless sizable benefits can be obtained from caching the value in the database. Just be smart about it.

To sum up - you should use the Transients API for values that take so long to compute that fetching them from the database is likely to always be faster. You should definitely not use the Transients API to cache a large number of entries, such as one or multiple transients for every user, as this may bloat the database on larger sites. For operations that are likely to be similar or faster than querying the wp_options table - use Object Cache. Whatever you do, make sure

you analyze and understand the implications of your choice. The right caching solution is dependent on context and should be determined on a case-by-case basis.

4. Full-Page Caching

Full-page caching is the process of caching and serving the response body. Imagine that WordPress had to run its entire request lifecycle for every request, including all hooks, functions, etc., just to render the exact same HTML for every user. That's insanely wasteful and inefficient. Full-page caching solves that by generating this response only once, storing it, and serving it like a static file. It can make your website hundreds of times faster.

There are 3 different types of full-page caching:

- PHP-level full-page caching
- Server-level full-page caching
- Reverse Proxies & Edge Caching

PHP-Level Full-Page Caching

PHP-level full-page caching is always dependent on a WordPress plugin. This plugin almost always creates a /wp-content/advanced-cache.php file and defines a WP_CACHE constant in wp-config.php. The advanced-cache.php file is a drop-in file, just like object-cache.php that we covered a moment ago.

If this file is present (and WP_CACHE is defined), it will be loaded very early in the request lifecycle. It is included at the very beginning of wp-settings.php, pretty much right after wp-config.php is loaded. This means that it will run way before any theme, plugin, or even most core code (including the database connection).

It is designed that way to circumvent as much PHP processing as possible when a cached version of the page exists. This is the job of this file. It looks for a cache of the requested page. If it is found, it renders it and calls die(). Execution is stopped, WordPress isn't fully bootstrapped, and a lot of time and processing is saved.

Remember that this drop-in's logic is specific to the plugin you choose. Many plugins have different ways of storing and finding the cached page. The most typical is storing the cache as an HTML file on disk, but there's nothing stopping you from storing it in persistent Object Cache. These plugins can usually be configured to fit your needs.

But what if the cache isn't present? In that case, normal execution continues. But there's a catch - if the current page can be cached, the plugin will capture the rendered HTML and cache it. This way, only the first user has to go through the lengthy PHP processing, and all other users receive the cached version. The way this process works is an implementation detail, but I can imagine those plugins buffering the output with ob_start().

The biggest advantage of PHP-level caching is that it works on all hosts - independent of the environment. It also theoretically provides a little more dynamism, as PHP runs before the content is returned. Unfortunately, it is the slowest out of the three.

Server-Level Full-Page Caching

Another strategy utilizes the web server's rewrite rules. This also requires a WordPress plugin. It basically works like PHP-level caching, except it usually doesn't use the advanced-cache.php drop-in. Instead, it generates .htaccess rewrite rules (or other for different web servers) that make it so that index.php is never hit if the cached file exists.

As you can imagine, this is even more performant than PHP-level caching. The WordPress request lifecycle never even begins. Your .html cache file is served to the user the exact same way a static image is. No PHP processing happens, which can be both a blessing and a curse, depending on your requirements.

Many plugins support both server-level and PHP-level configuration. One such example is W3 Total Cache, which has options for both. Some don't even bother with PHP-level caching. WP Fastest Cache seems to be one of them.

The main advantage of this strategy is speed. PHP is never run, which makes the response faster. That being said, a plugin like that might not work on all hosts. Rewrite rules are specific to the web server. If a plugin only modifies .htaccess and your website is run with Nginx with no Apache, the rewriting won't work. This can sometimes be problematic, especially on shared hostings or other environments that you don't control.

Reverse Proxies & Generic Server-Level Full-Page Caching

This type of full-page caching is the only one on my list that doesn't require a WordPress plugin. A reverse proxy is a piece of software that sits between the user and your web server. One popular example is Varnish. Its job is to intercept requests from the user and serve the cached page if it exists. The request never even reaches your web server. Content Delivery Networks (CDNs) would also qualify under this section.

Another side of this coin are caching modules built into web servers. Pretty much all web servers have them, i.e., Apache, Nginx, LiteSpeed, etc. If configured, they can cache the responses from PHP and serve them without ever having to consult the rewrite rules.

These solutions are completely detached from WordPress. It has its benefits - they are usually faster and more powerful. That being said, the downsides are stark. First of all - because they are WordPress agnostic, they aren't preconfigured to work well with WordPress. This means that in most likelihood, you will have to manually configure them to not cache pages for logged in users or other sensitive URLs - configuration that is almost always natively supported in WordPress caching plugins.

Another big problem is purging the cache. Caching plugins can hook into different actions and update the cache, e.g., on post edit, when a new post is published, etc. When using a reverse proxy, it has no idea about any of that. You have to make sure that the cache is purged when you need to purge it. In reality, this is usually achieved by installing a plugin that forces the purge on certain events (Proxy Cache Purge is often used for Varnish).

Solutions like that usually require more advanced system configuration and knowledge. If you mess it up, you're risking quirky behavior (website not updating on edits) and data leaks (personal information getting cached). One exception and an interesting case study is LiteSpeed Cache.

The LSCache plugin is a WordPress plugin developed and maintained by the official LiteSpeed team. It is basically a connection layer between your website and the LiteSpeed web server. It takes care of all the WordPress specific things, while the actual cache is stored and served by the web server. It doesn't cache the files itself - it intelligently communicates with the web server to manage its cache. The benefit is blazing fast performance with built-in WordPress integration. I can recommend it if you're using LiteSpeed as your web server.

Which Full-Page Caching Strategy To Use

All of these three caching strategies have their pros and cons. PHP-level caching is the most versatile of them all. It usually doesn't require any configuration and just works out of the box on all hosts. The price you pay for that is worse performance. Avoid this strategy if you can.

Server-level caching is usually the perfect compromise. It's faster than PHP-level because it completely circumvents any PHP processing by serving the static HTML files with custom rewrite rules. That being said, it's not without downsides. The one you're most likely to encounter is problems with rewrite configuration. This is especially true if you're using a web server other than Apache, like Nginx as the sole server.

Reverse proxies and generic web server cache can be very powerful in certain contexts. Varnish is probably more capable and robust than any WordPress plugin. CDNs can allow you to introduce edge caching for truly blazing-fast performance. The price you pay is complexity.

Reverse proxies should absolutely not be used by hobbyist bloggers with 5 blog posts. They require much more technical configuration and testing in order to make sure everything works as expected. For most WordPress websites, it's like using a sledgehammer to crack a nut - a good server-level caching plugin will definitely suffice. That being said, if you're qualified and are willing to spend more time on configuration, they can make a very big difference (especially CDNs like Cloudflare). The exception is obviously LiteSpeed Cache which I already discussed.

Caching Rules

Full-page caching for some static websites can be set up by just installing a plugin and forgetting about it. Unfortunately, most websites aren't like that. Making sure the right pages do

and don't get cached is crucial for making your site work as expected. I'm talking specifically about establishing correct caching rules. Caching rules are exactly what you think they are - they specify which pages should and should not be cached.

Query Strings

One important area is query strings. Which query parameters should be cached, and which should be ignored? Should URLs with query strings be cached at all? These aren't simple questions.

Let's take a request to /blog/?_ga=123googleanalyticsidentifier. Should it serve the page that was cached for /blog/? Yes, because the query parameter doesn't change the rendered HTML. On the other hand, let's take /blog/?search=post. Should this be served the blog cache? Absolutely not. The rendered HTML will be completely different (assuming it's a search result for "post").

In that case, you have to configure your caching solution to ignore "_ga". Some caching plugins will automatically create separate cached versions for every not-ignored query parameter (i.e., our "search"). That being said, many plugins default to not caching query strings at all. A request to /blog/?search would always run the full WordPress PHP execution.

There's a good reason for that. Query strings are usually used for rendering dynamic content, exactly like the search functionality. Unless you know what you're doing, it is safer not to cache those pages. The number of cache entries can also explode. /page/?foo=bar would create an entry, and so would /page/?foo=baz. If you store cached pages indefinitely or with a very high TTL, a malicious user (or even bot traffic) could fill your disk space with cache files.

If you know what you're doing and what query parameters your website expects, you may see a significant performance boost from intelligently caching and purging those. This part is completely yours to configure. No one knows what pages and parameters are present on your website better than you do. Make sure to set the correct rules and test it ruthlessly (including purging on different actions). Just keep in mind the risks that go along with caching query strings.

Cookies

Cookies are similar to query strings in that their value may change the rendered content. Take a language cookie as an example. A multilingual plugin can set a cookie to "remember" the language of the website the user selected. In such a case, the presence of the cookie results in different dynamic content being rendered. Another example could be a multicurrency WooCommerce plugin.

Pretty much all of the same tips and caveats apply to cookies as to query strings. Most caching solutions allow you to completely skip cache (i.e., serve uncached content) when a certain cookie is present. This is how these plugins skip cache for logged in users (which is the default

behavior for pretty much all plugins). Some of them allow you to create different caches for different values of the cookie. The same problems as when caching query strings apply here (a large number of files, configuring the right cookies, purging, etc.).

Pages & Other

Naturally, pretty much all caching solutions allow you to specify URLs that shouldn't be cached (usually allowing wildcards like /blog/* to not cache any blog posts). This is useful when you have a few pages that are very dynamic and you want to always serve them uncached.

There are also a plethora of other possible cache rules to configure, such as caching by user agents, post categories, or even custom fields. Different plugins allow for different customization. Whatever you do, make sure to be smart about it. Make sure you understand what it means that a given page will or will not be cached and test your website in incognito. You should be well equipped by now to make knowledgeable decisions.

Dynamic Content

We've actually already discussed caching with dynamic content briefly when talking about nonces in the security chapter. Let's start by covering logged-in users, as this is usually the biggest pain point. As I already said, the typical configuration is to completely skip cache when the wordpress_logged_in_* cookie is present. This solves the problem, but is also the most inefficient.

Another option that some plugins provide is a separate cache for each user. This solves the problem of personalization. If the user's name is shown on the page, it will also be present in their cached static file. While this approach might seem attractive, it's very problematic. First of all, there's usually little to no benefit from that. Logged-in users won't typically visit the same page multiple times. Even if you cache it, they will only request it one time - and the cache won't exist yet.

Secondly, if you have many users, the amount of cached pages can grow quite fast. This is also the case with query strings and cookies-varied cache (as all of them are basically just dynamic pages), but caching logged-in responses only adds fuel to the fire. Also, remember that the data of the user is not available until way after advanced-cache.php was run. This means that logged-in cache will likely be less efficient than even typical PHP-level caching (unless the plugin handles that with proprietary logic).

Overall, the general consensus, not only in the WordPress community, but also in the broader web development community, is to skip full-page caching for logged-in users. A logged-in experience is, by its nature, dynamic. You will be much better off trying to optimize your code by utilizing Object Cache and transients where applicable (and by getting rid of heavy, inefficient plugins).

That being said, there is technically a legitimate way to still utilize (almost) full-page caching for logged-in users and pages with dynamic data. It is only really realistic if you have full control over the entirety of the website, particularly over the theme (preferably custom).

I'm talking about making your PHP-rendered HTML not contain any personal/dynamic data. Instead, it would contain placeholders, and the actual content would then be loaded using AJAX on the frontend or with Edge Side Includes if your caching solution supports it (LiteSpeed Cache does!). This approach is sometimes referred to as hole punching.

With an architecture like that, you could serve the same static HTML file to all of your visitors. Although possible, you should ask yourself if it's realistic. It will probably not be compatible with most plugins and will require a lot of custom code. I could see it being used for certain high budget projects, but 99.9% of WordPress websites will never want to do this.

5. Browser Caching

The very last layer in caching is the user's browser. Most browsers will cache pages by default. Browser caching is controlled with response headers sent by the server. There are a few different headers that have been used for different purposes over the years, but the most important modern header is Cache-Control.

Cache-Control allows you to specify directives like max-age, no-cache, private, public, stale-while-revalidate, stale-if-error, etc. These directives change the behavior of the clients. As a matter of fact, they aren't only used by browsers. They are usually the way the web server communicates with reverse proxy caching solutions, such as Varnish or Content Delivery Networks.

This topic is way outside the scope of this guide. Besides, as a WordPress developer, it's rather unlikely that you will ever have to interact with these headers directly. Go read a little about Cache-Control to have this knowledge in the back of your head. It may come in handy when you encounter some obscure problem that should not have the right to exist (which is usually the type of problems caused by cache).

Sending Emails

Every WordPress website has to send emails. I'm not talking about marketing or newsletter emails. I'm talking about useful emails, like resetting a password, a registration notification, or a WooCommerce order confirmation. When emails sent by your applications don't get delivered, you can safely consider it broken. Yet, many WordPress users have a really hard time with email deliverability, and there are good reasons for that.

Let's take a step back. There's one central function in WordPress that lets you send emails - wp_mail(). This function expects 5 arguments:

- **$to** - the recipient's email.
- **$subject**
- **$message**
- **$headers** - email headers, like "From", "BCC", "CC", etc.
- **$attachments** - paths to files on the disk that are to be included as attachments.

Internally, this function uses the PHPMailer library to format and send the email. By default, the email is sent with the PHP mail() function. This function passes this email to the Mail Transfer Agent (MTA) installed and configured on your server. An MTA is just software that manages email transfer, kind of like a digital postman. Some examples include Postfix, Exim, Sendmail, or Microsoft Exchange. What MTA you use is fully dependent on your server's configuration - it doesn't have anything to do with WordPress.

The Global Mail System

How the email system works is a little beyond the scope of this guide, but unfortunately it's critical to understanding this topic, so I'll give you a brief overview. As I already said, emails are sent by MTAs. To do that, they use the Simple Mail Transfer Protocol (SMTP). The way this works is your MTA connects to your recipient's MTA and transfers the message.

The problem is - anybody can send an email saying that they are someone else. The recipient MTA has no way of knowing that the email was actually sent by johndoe@example.com. This problem is solved by two DNS records - SPF and DKIM. SPF is basically just a list of IP addresses that are allowed to send an email in the name of your domain (example.com). DKIM is a cryptographic scheme allowing the receiver to verify that the message was signed with your private key (which only you, that is - your MTA, has access to).

What the receiving MTA does with this information is completely up to its configuration (actually, it also depends on DMARC, but that's mostly irrelevant here). It can be lenient and allow all messages in, even if the sender's IP isn't present in SPF and DKIM isn't used. That would, however, be very insecure. Most MTAs will either send mail like that to spam or completely reject it. That's where I'm going to stop my explanation of the mailing system. If you still don't understand it - go ask ChatGPT about it.

Why mail() Often Fails

As already noted, the wp_mail() function uses the native PHP mail() function to send the message, which relays it to the local MTA. There are multiple reasons why this might fail. The most trivial one is that the web server doesn't permit mail(). Many shared hosting environments disable this function completely in order to prevent bad actors from sending spam and getting their IP blacklisted. In that case, the email will never even reach an MTA.

But let's say that your hosting environment allows mail() and has an MTA configured. Even if your website's domain name is example.com and you send an email as

johndoe@example.com, there's a good chance that your mail server is different from your web server. If that's the case, your web server's IP will most likely not be present in your SPF (unless you added it), and your web server's MTA will definitely not have the private key required for DKIM verification (unless you configured it, which isn't usually possible on hosting environments prepared for websites). If the recipient's server requires those, your message will be rejected.

The mailing system is complex and there are probably many more possible causes for mail() not to work. One of them is the fact that some sending MTAs don't allow sending emails if the actual email matching the "From" header doesn't exist. Here's the catch - if you don't supply this header yourself, WordPress will default to wordpress@example.com (assuming example.com is your website's domain). If this email doesn't exist, an MTA with those limitations will refuse to send the message.

Making wp_mail() Reliable

I think you should understand by now why using the default mail() is likely a bad idea. I mean, you could add your web server's IP address to your SPF record and configure DKIM on your MTA, but that's making your web server be your mail server - you already have a mail server for mail. That's of course unless your web server already *is* your mail server. In that case, you don't really have to configure anything more.

On most websites, you'll have to set up an SMTP connection. The usual way of doing that is with a plugin. WP Mail SMTP is the most popular one but there are quite a few choices. What these plugins do is either hook into wp_mail() or override it completely (wp_mail() is pluggable). This way, they can modify the behavior of PHPMailer so that it doesn't try to send the email directly to the receiver but instead relay it through your chosen SMTP server.

Your job would then be to configure the plugin with all the details needed to make an SMTP connection to your actual mail server, i.e., host, port (usually 465 or 587), username, and password. When using SMTP, instead of WordPress trying to send your email directly, your web server will create an SMTP connection to your mail server. Your web server will authenticate with your details (like you just logged into your email account) and will pass the message to your mail server. Your mail server will then send it to the receiver.

This way, the email is actually being sent by the MTA on your mail server, not your web server - exactly as if you logged into your email account and sent it manually. Assuming you've correctly configured your mail-related DNS settings, the message should have far fewer deliverability problems. You're also not using mail() anymore, which means that you can send emails even on servers that disable it.

Some of these plugins also include built-in support for popular transactional email providers, such as SendGrid, Postmark, Mailgun, Amazon SES, and more. These are external email providers that you can configure and use (for a fee). They ensure superior deliverability

compared to most self-hosted mail servers. Look into them if email is an important part of your website (e.g., on a WooCommerce store).

Modifying wp_mail() With Hooks

There are a few useful filters inside wp_mail() that allow you to modify its behavior. Search for hooks with "wp_mail" or just look at the function's source code. Some of the interesting ones include wp_mail_from, wp_mail_from_name, wp_mail_charset, and wp_mail_content_type. For example, you could change the content type of your email from text/plain to text/html like this:

```
function pgn_set_html_content_type() {
    return 'text/html';
}

function send_my_html_email() {
    // Add the filter right before calling wp_mail()
    add_filter( 'wp_mail_content_type', 'pgn_set_html_content_type' );

    $to = 'user@example.com';
    $subject = 'This is an HTML email';
    $message = '<html><body><h1>Hello, World!</h1><p>This is a
paragraph.</p></body></html>';

    wp_mail( $to, $subject, $message );

    // IMPORTANT: Remove the filter immediately after to avoid affecting other emails.
    remove_filter( 'wp_mail_content_type', 'pgn_set_html_content_type' );
}
```

Feeds

There is one URL path on every WordPress website that most people don't know about - /feed/. A feed, like an RSS feed or an Atom feed, provides an alternative way of reading content. These are basically just endpoints returning all of your posts in a structured format (XML).

Feeds used to be much more popular in the early days of the internet. They are the foundation of what's called web syndication. Imagine you regularly read 10 blogs. Normally, you'd have to visit every one of those blogs every day to check if there are any new posts. But if you were a little more tech savvy, you could use a feed reader and give it a link to the feed from each of those blogs. If you did that, the feed reader would check for new posts automatically and would even allow you to read them in the reader itself, with a unified interface.

The default /feed/ path is an RSS 2.0 feed, but there are a few more. For example, the path /feed/atom/ contains the same post information but using the Atom feed structure. Feeds are also available for other content types, like comments (/comments/feed/), categories (/category/sci-fi/feed/), etc. I'm not going to include a feed example as it's a little verbose. Google it (or check it on your own WordPress install) to see it for yourself.

WordPress Multisite

Multisite is an advanced WordPress feature. It lets you create multiple "subsites" under the same website (using the same core files and database). It is useful for enterprise-level purposes with multiple teams having to manage their own part of the entire website, e.g., universities, franchises, SaaS products, large blog networks, etc.

Multisite originated from the separate WordPress MU (multi-user) project. It got merged into core in WordPress 3.0. This is why some old functions and hooks are prefixed with "wpmu_". The two most important terms in Multisite are "site" and "network". A site is a single site instance with its own content, uploads, etc. A network is the set of all sites. Every site is managed by a site administrator. The network is managed by a network administrator (or super admin).

Multisite works by modifying the database and filesystem structure. Every site in a network has its own content tables (wp_posts, wp_postmeta, wp_terms, etc.), but there are also network-wide tables. All files except for the uploads directory are shared. This means that all sites run the exact same plugins, themes, and core.

Plugins and themes can only be installed and updated by the super admin. Site admins can only choose to enable or disable plugins made available (installed) by the network administrator, unless the network administrator has made them required for all sites. Site admins can also choose to activate one of the themes installed by the super admin.

By default, a Multisite install can be either subdirectory or subdomain-based. A subdirectory-based Multisite exposes individual sites at subdirectories, i.e., example.com/site1. Subdomain-based Multisite does so on subdomains, i.e., site1.example.com. Note that all of these use the same domain (example.com). Since WordPress 4.5, you can modify a single site to use a completely separate domain. This process is called domain mapping.

Database Structure With Multisite

When you configure Multisite on your WordPress install (which we'll soon do), a few new tables get added to your database. New network-wide tables are:

- **wp_site**
- **wp_sitemeta**
- **wp_blogs**
- **wp_blogmeta**

- **wp_registration_log**
- **wp_signups**

wp_site and wp_sitemeta contain information about networks. Theoretically, a single WordPress Multisite installation can have multiple networks. In practice, this is never used and it's not supported by default in WordPress (there's no UI for that). wp_blogs and wp_blogmeta contain information about single sites in a network.

Why is it wp_site and not wp_network? That's because of Multisite's history. A network used to be called a site, and a site used to be called a blog. What we now call "sites in a network" used to be called "blogs on a site". That's something to keep in mind to not get confused with these naming quirks.

The following tables are specific to every site:

- **wp_posts**
- **wp_postmeta**
- **wp_terms**
- **wp_termmeta**
- **wp_term_relationships**
- **wp_term_taxonomy**
- **wp_options**
- **wp_comments**
- **wp_commentmeta**
- **wp_links**

Every site in the network gets its own set of these tables. They are named with the ID of the site. The posts table for the site with ID 2 would actually be named wp_2_posts. If you added another site, it would get tables like wp_3_posts, wp_3_postmeta, etc. The first site (created before configuring Multisite) keeps the original tables without the ID.

Do you see anything missing on that list? What about **wp_users** and **wp_usermeta**? Well, I got a surprise for you - these are network-wide and shared across all sites. That's right, a user on one site is also a user on all other sites in the same network. This is so important that I've devoted an entire section to it down below.

File Structure With Multisite

The file structure for Multisite is the exact same except for the uploads directory. Every site has its own media library. The directory doesn't change for the original website (with ID 1) - the files still live under /wp-content/uploads. New sites' files are organized into site-specific subdirectories. The path is /wp-content/uploads/sites/ID/. A site with ID 2 would store media in /wp-content/uploads/sites/2/.

User Accounts With Multisite

This is perhaps the most interesting and confusing part of Multisite. As I already said, the wp_users and wp_usermeta tables are shared across the entire network. This means that an account created on one site can log into every other site in the network. But there's a catch - some meta values are scoped to the site by including the ID in the meta key, e.g., wp_2_capabilities.

A user that registers an account on site with ID 2 will have the wp_2_capabilities meta value containing the "subscriber" role. They will, however, not have any value for wp_3_capabilities (this meta key will not be created for the user, even if a site with ID 3 exists). This means that for the site with ID 3, they have absolutely no capabilities. If this site checks if current_user_can() anything - it will return false for this user.

This fact alone doesn't stop them from logging into the site and acquiring a session. /wp-login.php will indeed log them in, as it doesn't look at the capabilities but only at wp_users to see if the user with this login and password exists. But if they try to visit /wp-admin/ (which is something even subscribers can do to edit their own details) - they will get a message telling them they don't have the permissions to do that. They have a session, the global $user is populated, and is_user_logged_in() returns true, but they don't have any capabilities.

An even more interesting quirk is that site administrators only see users in the users list that have the wp_ID_capabilities meta key for their site. This means that someone can be logged into your site, but if you visit the 'Users' tab in the admin panel - you will not see them in the list of all registered users! This can be particularly confusing if your site offers functionality that doesn't require any capabilities. The user will be able to perform that action and you still won't be able to see them in the list - like a ghost.

I was curious to see how that worked and set up an experiment. I installed and activated WooCommerce for my entire network. I logged into example.com/site1/ using an account that doesn't have any capabilities on that site. I headed to /my-account/ and modified my account details. I logged into the main site on example.com with the same account and viewed my details. Sure enough - the details I saw were the ones I set up on the site I didn't have any capabilities on!

Turns out WooCommerce stores those details in wp_usermeta with the "billing_first_name" and "billing_last_name" keys (and they also update the default "first_name" and "last_name" meta values). Those aren't scoped to the site with the site ID! They also don't check for any capabilities on pages that allow you to change your own details.

This means that if you're running WooCommerce on Multisite, user details set up on one site will be used across the entire network. This is also the case with any other plugin that uses the general built-in user details (e.g., first name, nickname, email, etc.) or any other user metadata that isn't scoped to the site. That's certainly something to keep in mind.

PHP Multisite Functions

There are a couple of important Multisite functions and classes you'll use when writing plugins. The single most important one is probably **is_multisite()**. This function returns either true or false, depending on if multisite is enabled.

There's also a peculiar function **wp_is_large_network()**. It's used to check if the current network is considered "large". The default criteria for a large network are either 10,000 sites or 10,000 users (depending on the argument you pass - default is 'sites'). These numbers can be changed with filters. This function is used in a few places by core - probably for optimization purposes.

One of the most important functions is **switch_to_blog()**. This function lets you switch the context from the current site to another site in the network (you pass its ID). The only two things it does underneath is change the database prefix (i.e., from "wp_2_" to "wp_3_") and change the Object Cache prefix (some Object Cache entries are automatically scoped to the site by prefixing the key with the blog ID). You have to remember to always call **restore_current_blog()** afterwards, which reverts these changes.

switch_to_blog() isn't very useful unless you couple it with something like **get_sites()**. This function lets you query for sites. You can, for example, fetch a list of all site IDs in the network and iterate over them. Inside the loop, you can call switch_to_blog() and display some information about this blog with get_bloginfo() or some posts with WP_Query or get_posts().

You can also create a new site programmatically. **wp_insert_site()** inserts the site and its information into the wp_blogs table. After this is done, you have to call **wp_initialize_site()**. This runs the initialization routine for the site - including creating all the database tables. You can use those if you want to create a custom UI for adding sites by users (useful for SaaS).

get_site_option() and **update_site_option()** let you get and update options stored in wp_sitemeta. This is the network-wide metadata table. It really functions like wp_options but for the entire network. Since WordPress 4.4, these functions are wrappers over **get_network_option()** and **update_network_option()**. These are more semantically correct - you can use them directly if you're not planning on supporting <4.4. They default to get_option() and update_option() if multisite isn't enabled, so you can safely use them without checking if is_multisite().

How Other Functions Are Affected By Multisite

Multisite is probably the single most important reason why you should always use the built-in WordPress functions and APIs. You hard-coded the path to /wp-admin/ in your code? Guess what - if someone sets up a site at example.com/site1/, their wp-admin will live at example.com/site1/wp-admin/ and your code will break. That's exactly why I've been preaching the use of built-in URL and path functions this whole time. They take care of that for you.

Naturally, all database-related functions are indirectly affected by Multisite. This is because the global $wpdb object has the table prefix scoped to the site. Yet another reason to use the built-in APIs. The Object Cache also scopes your entries (the cache key) to the site, unless the group you're adding the entry into is global. You register a group as global with wp_cache_add_global_groups(). If that's the case, the cache key will be the same across the entire network. Some default global groups are users, user_meta, theme_files, sites, networks and more.

When To Use Multisite

Multisite shines when:

- you want one WordPress installation to manage multiple sites with shared code (plugins, themes, and core),
- you want shared user accounts across sites,
- the sites are intrinsically linked together (one displaying content from another),
- you want to scale site creation (deploying a new site with the same theme and plugins is much easier than installing and configuring a new WordPress instance).

Multisite is great for when you have many subsites that are somehow related to each other. Imagine a university website, where each department needs to have their own subsite. You can also use it for some SaaS products. WordPress.com is probably the largest multisite network out there. It's a commercial managed hosting service run by Automattic. Every website hosted on WordPress.com is a site in the network (although it's far from default Multisite because of its scale).

Multisite breaks down when:

- the websites have wildly different requirements,
- you need isolation of data and resources,
- the individual site admins need to have full control over their websites (installing plugins, themes, etc.),
- you're not sure if Multisite is a good idea.

In general, Multisite is not a good idea. Use it only if you've analyzed your case and decided otherwise. It adds complexity and reduces robustness. First of all, you have one database that houses all the data. What if one site gets hacked? The hacker has access to the entire database - including all the data from the other sites. The same thing is obviously also applicable to files on the disk. This is the main reason why it's usually stupid to use Multisite to host websites for different clients.

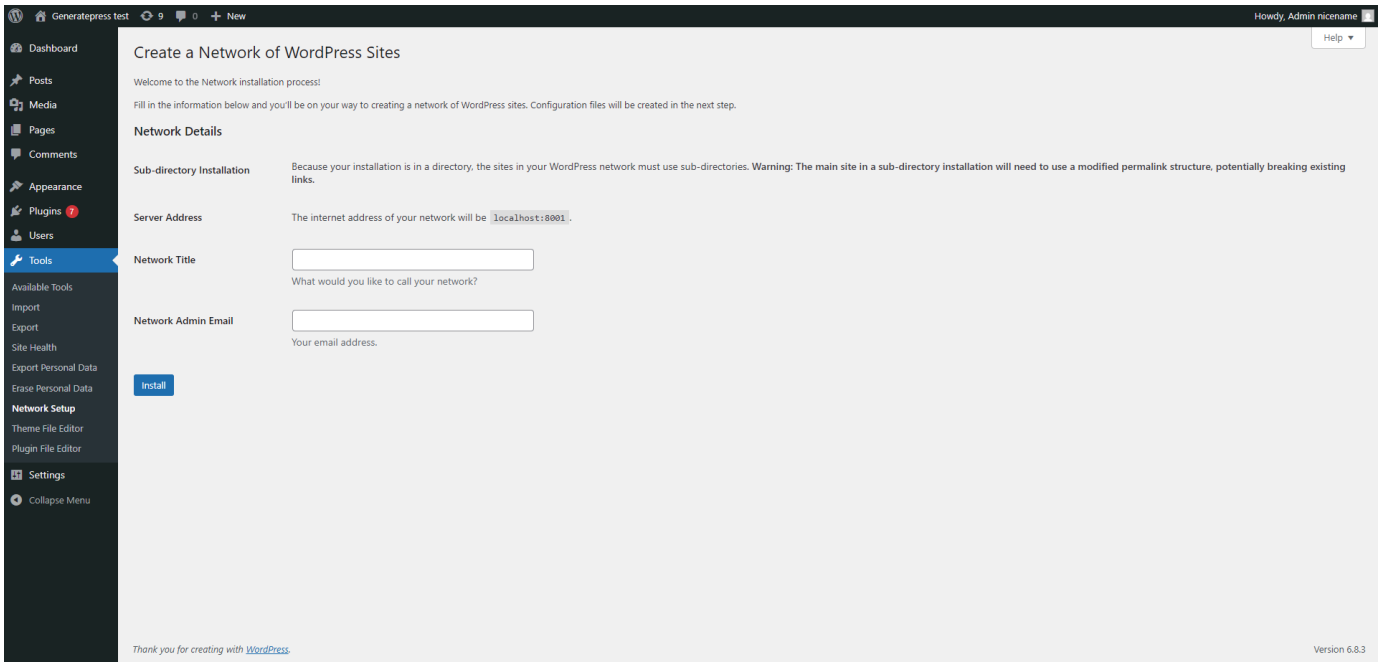
The resources are shared across the entire site. If one site gets DDoSed, the entire network goes down. If one site hogs the database connection, no other site in the network can reach the database. The fact that you're hosting multiple websites with the same database inherently increases the stress it's going to have to endure. Don't choose Multisite if you can't afford to have that happen.

If you need vastly different plugins on different sites in the network, it's usually a sign that you shouldn't be using Multisite. You're going to have to install all of the plugins for the entire network. The individual site admins will then pick the ones they want to enable. Can you imagine being the site admin and having 100 inactive plugins because they are needed by other sites in the network? And let's not forget that a vulnerability or a critical error in one plugin can take down the entire network.

Configuring Multisite

The following is just a quick overview of the process of configuring (and managing) a Multisite installation. Read the official WordPress Multisite documentation before you do it yourself.

Multisite can't be enabled when installing WordPress. It can only be activated after the fact. To do it, you have to add "define('WP_ALLOW_MULTISITE', true);" to your wp-config.php. When you do that, a new menu tab called "Network Setup" will appear under Tools. Here's what you'll see when you click it:

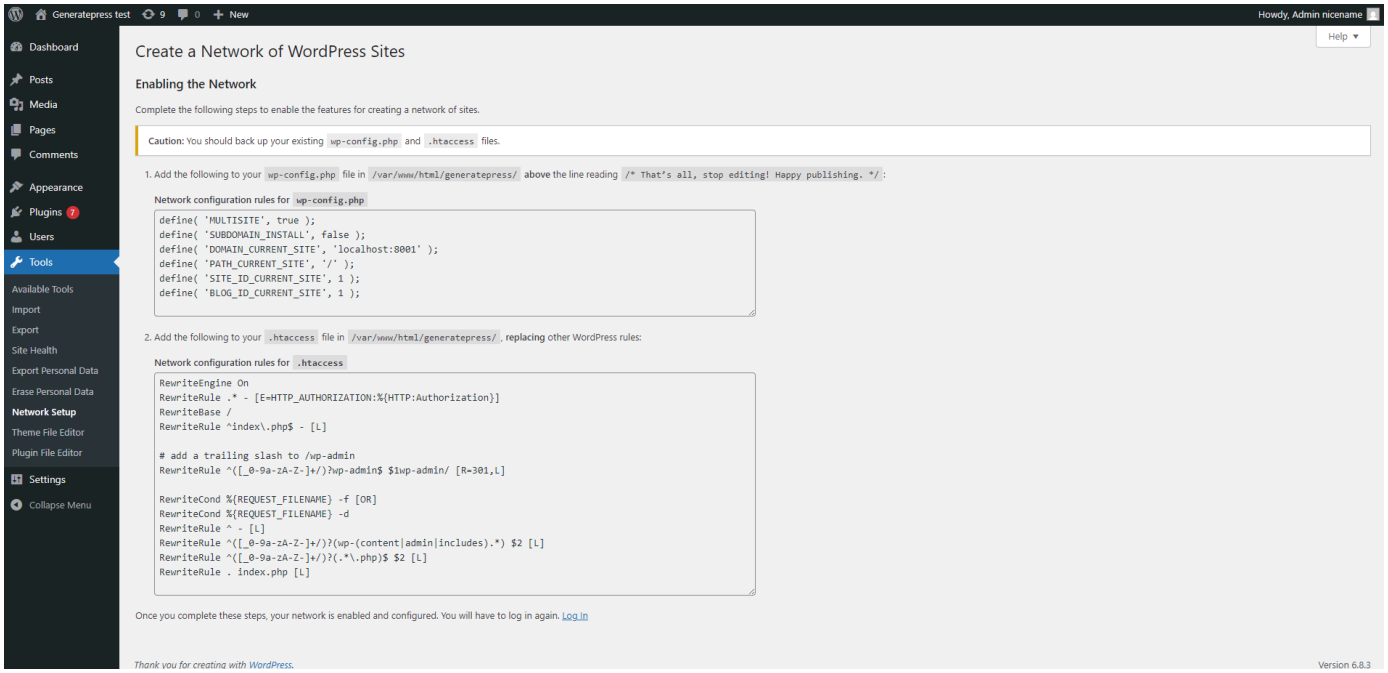


As you can see, you're given the choice to configure the name of the network and the email address of the super admin. This screenshot is from a localhost installation which forces the use of subdirectory. If it was a typical website, you'd also be able to select between a subdirectory and subdomain.

Note that, if your website is over 1 month old, you will be forced to select subdomain to prevent clashes between your new site directories and existing pages. If you're thinking "this is stupid" - I'm with you. The absurdity of measuring the likelihood of URL clashes with time since

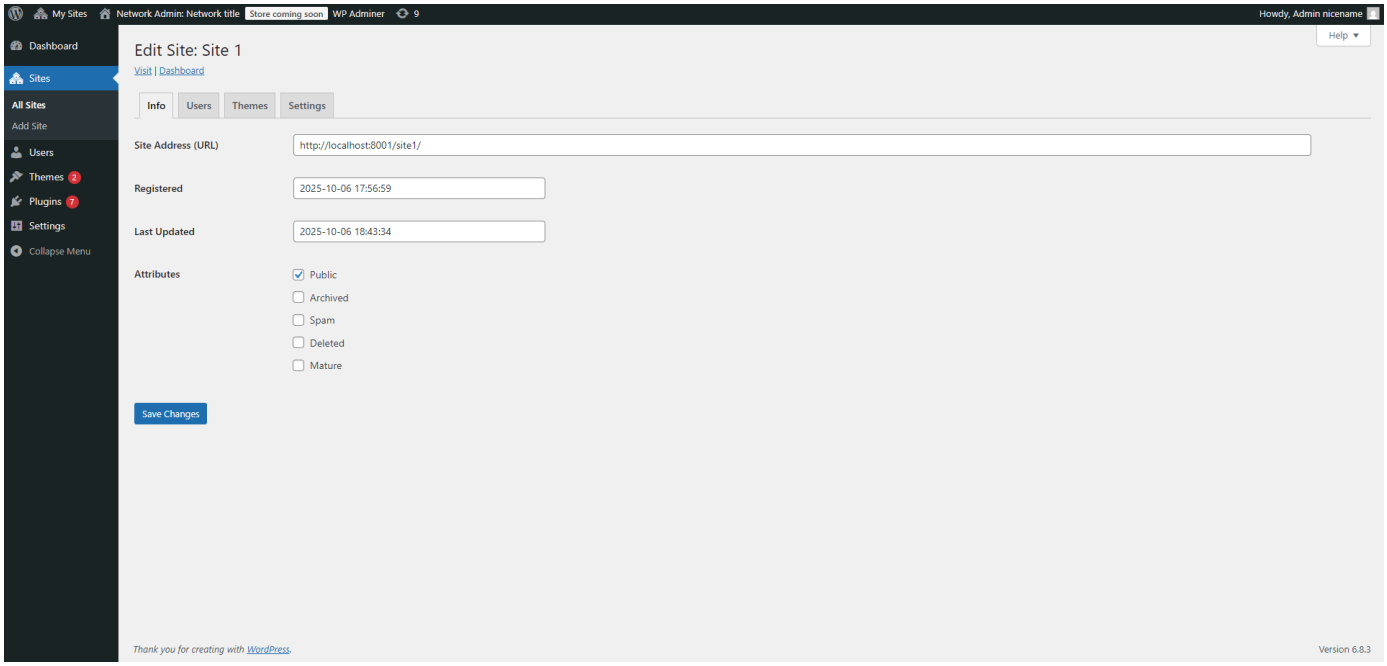
WordPress was installed is incomprehensible to me. Thankfully, you can change it later (at least the documentation says so).

After you click "Install", WordPress will give you code snippets you have to paste in wp-config.php and .htaccess. After doing that, Multisite will be enabled.



You will immediately see that your dashboard changes. The most important thing is that you'll be redirected to example.com/wp-admin/network/. This is the URL under which you can manage your entire network. If you visit example.com/wp-admin/, you will get back to the site-specific dashboard for your "original" site (with ID 1 in the network).

You can add sites by navigating to Sites > Add Site, manage users by navigating to Users, enable plugins or themes in Plugins or Themes, and configure network-wide settings in Settings. Every single site has four tabs: Info, Users, Themes, and Settings. You can use them to configure this site as a super admin. A typical WordPress admin dashboard is available to site admins (and you) under /site1/wp-admin/.



PS: If you choose subdirectory instead of subdomain, your posts permalinks structure will suck. It will be prefixed with "/blog", so you'll get /blog/post-name/ instead of just /post-name/. I've seen some solutions online, but I wasn't able to get them to work in WordPress 6.8.3.

Plugin Compatibility With Multisite

Not all plugins are compatible with Multisite. Similarly to security, the biggest reason for incompatibility is lack of competence. Some plugin developers don't know what Multisite is or how it works, and they don't even know that they need to make their plugins compatible. That's why you should pay attention to this chapter, even if you aren't planning on enabling any Multisite installations.

Most WordPress functions just work out of the box. This is particularly true if you're using built-in APIs. I've already addressed how functions' behaviors change with Multisite. The biggest problems with plugins on Multisite are logic-related. When writing your plugin, you have to constantly keep asking yourself "how should this work on Multisite?". I'll name a few examples you should look out for, but this list will not be comprehensive. Do your own reading and thinking when writing a plugin.

Let's start with one of the most common requirements from a plugin - a menu page. Remember how we added a custom menu page to the admin panel to modify the plugin settings? How should that work on Multisite? Should the page be available in the networks dashboard or in the site dashboard? You have to think about it and use the appropriate hook to add the page.

Let's assume you want the menu to be added only to the network dashboard. You should probably set the "Network: true" header in your plugin's main file to make it impossible for site

admins to enable it. But if you're doing that and you want your plugin's configuration to be global to the entire network, then you should store the settings in wp_sitemeta. Well, guess what, the Settings API doesn't support that and will store it in wp_options of your site with ID 1. This will make them inaccessible by default on other sites. You have to deal with that.

Let's say you figured that out. Your options are stored in wp_sitemeta. Now you have to remember that you can't use get_option() to fetch them. You have to use get_site_option() to get and update_site_option() to update them.

But what if your plugin was activated and configured before Multisite got enabled? I'm assuming you fell back to the normal admin menu page when is_multisite() was false. Your configuration is stored in wp_option, and suddenly, the website administrator enables multisite. Your get_site_option() calls worked before (since they default to get_option() if multisite is disabled), but now they're checking wp_sitemeta which is empty. Guess what? You have to deal with that yourself :)

And did I mention custom database tables? That's where real fun starts. Let's say you created a table named "{$wpdb->prefix}table" on the activation hook. The prefix will be scoped to the site. When a site admin on site with ID 2 activates this plugin, a table called "wp_2_table" will be created. Every site will get its own table, which you might or might not want.

But there's a catch - what if the plugin is network-activated (by the super admin, enforced for all sites)? The activation hook will run only once, and in the context of the site with ID 1. Only one table will be created - "wp_table". Even worse, your code that queries "{$wpdb->prefix}table" will only work on the main site, as all other subsites will try to query "wp_{ID}_table", and this table will not exist. If you need site-scoped tables, you have to manually iterate over all sites and create them in the activation hook if it's activated network-wide (this information is passed as an argument to the activation hook).

And even that is not enough. What if the super admin adds a new site after they've already network-activated your plugin? The activation callback will not run for this site and its table won't be created. You have to hook into wpmu_new_blog and create it. By the way, use $wpdb->base_prefix if you want the table to be global (not scoped to the site with its ID).

There are probably other edge cases you'll come across as you develop your plugins. The most important thing is to stay vigilant and constantly analyze your logic and how it should work on Multisite (unless you just say "screw it" and not support it). Oh, and test your damn plugins! Multisite can be the source of many weird bugs.

WP-CLI

WP-CLI is the Command Line Interface for WordPress. It is used to programmatically manage WordPress websites (from the terminal). It is not a part of core WordPress. It is an external open source project, although developed by many companies that work on WordPress, such as

Automattic. Its goal is to be the CLI wp-admin. Everything you can do in /wp-admin/, you can do with WP-CLI (and much more). Moreover, it is faster, more robust, and can be automated (e.g., in CI/CD pipelines).

WP-CLI is large. It has hundreds of commands and options. You could probably use it to manage your entire WordPress website without ever visiting /wp-admin/ in your browser. As such, it would be impossible for me to cover it in full. This would be an undertaking for a whole new guide, and I'm sure you already can't wait to finish this one (so can't I). Check out the official WP-CLI Handbook to learn more.

There are many ways you can install WP-CLI, but there are two primary ones. The most common is downloading the wp-cli.phar file, making it executable, and moving it to your PATH (with the name "wp"). When you do that, you can just use "wp {command}" anywhere on your system. The other one is using Composer. In such a case, WP-CLI is bundled with your project. This is not a tutorial so I won't show you how to do it. Go read the docs.

Commands

I feel like the most reasonable way to begin this chapter is by discussing a few useful commands. We can start with a simple example:

```
wp --info
```

This will show you some relevant system information, including the PHP version, location of the used php.ini file, OS version, WP-CLI version, etc. As I said, there are *a lot* of commands in WP-CLI. I will only list a few I think are the most useful, but I won't really explain them. You can find all that info in the docs. The WP-CLI Commands list contains detailed documentation of every command. Always look there when using WP-CLI.

wp cache - manage Object Cache:

- wp cache add
- wp cache delete
- wp cache flush
- wp cache get
- wp cache set

wp cap - manage capabilities of user roles:

- wp cap add
- wp cap list
- wp cap remove

wp comment - manage comments:

- wp comment approve

- wp comment create
- wp comment delete
- wp comment get
- wp comment list

wp config - manage and read wp-config.php:
- wp config create
- wp config edit
- wp config get - gets the value of a specific constant or variable.
- wp config set - sets the value of a specific constant or variable.

wp core - manage WordPress core:
- wp core install
- wp core multisite-install
- wp core update

wp cron - manage WP-Cron:
- wp cron event delete
- wp cron event list
- wp cron event run - you can use "wp cron event run --due-now" in a system cron job to make cron reliable.
- wp cron event schedule

wp db - manage the database:
- wp db create
- wp db drop
- wp db export
- wp db query
- wp db tables

wp eval - execute arbitrary PHP code. For example: *wp eval 'echo WP_CONTENT_DIR;'*

wp eval-file - load and execute a PHP file. For example, *wp eval-file ./path/to/wp-script.php*

wp i18n - manage internationalization:
- wp i18n make-json
- wp i18n make-mo
- wp i18n make-php
- wp i18n make-pot
- wp i18n update-po

wp media - manage media files:
- wp media import
- wp media regenerate

wp menu - manage navigation menus (classic themes):

- wp menu create
- wp menu delete
- wp menu item add-custom
- wp menu item add-post
- wp menu location assign

wp option - manage options:

- wp option add
- wp option delete
- wp option get
- wp option update

wp plugin - manage plugins:

- wp plugin activate
- wp plugin deactivate
- wp plugin install
- wp plugin uninstall

wp post - manage posts:

- wp post create
- wp post delete
- wp post edit
- wp post meta add
- wp post meta delete

wp post-type - retrieve information on registered post types:

- wp post-type get
- wp post-type list

wp rewrite - interact with the Rewrite API:

- wp rewrite flush

wp role - manage user roles:

- wp role create
- wp role delete

wp scaffold - generate file structure and code for different things (this one is cool af):

- wp scaffold block
- wp scaffold child-theme
- wp scaffold plugin
- wp scaffold post-type
- wp scaffold taxonomy

wp search-replace - search and replace strings in the database (serialization-aware).

wp site - manage sites on Multisite:

- wp site activate
- wp site create
- wp site deactivate
- wp site delete

wp super-admin - manage super admin users on Multisite:

- wp super-admin add
- wp super-admin list
- wp super-admin remove

wp taxonomy - retrieve information on registered taxonomies:

- wp taxonomy get
- wp taxonomy list

wp term - manage taxonomy terms and term meta:

- wp term create
- wp term delete
- wp term update
- wp term meta add
- wp term meta delete

wp theme - manage themes:

- wp theme activate
- wp theme delete
- wp theme install
- wp theme update

wp transient - manage transients:

- wp transient delete
- wp transient get
- wp transient set

wp user - manage users:

- wp user create
- wp user delete
- wp user get
- wp user update

wp widget - manage widgets:

- wp widget add

- wp widget deactivate
- wp widget delete
- wp widget move

That's a lot of commands, isn't it? Well, it's probably about 20-30% of all core WP-CLI commands, and I haven't even listed any options you can use to configure their behavior. That's what I meant when I said it was a large topic. I chose the ones above just to show you roughly what's possible with WP-CLI. Again - read the docs when actually using them.

PS: Many commands support the --format option, allowing you to format the response as JSON or something similar. This facilitates scripting.

How WP-CLI Loads WordPress

There are two types of WP-CLI commands - the ones that require WordPress, and the ones that don't. I'm not really going to dive deep into the internals of WP-CLI. Go read the source if you're into that kind of stuff. The one thing you need to understand is that the method managing WP-CLI execution tries to execute the command if it's intended to be processed before WordPress loads. If the command requires WordPress, it proceeds to bootstrap it first.

The way WP-CLI loads WordPress is pretty interesting. It doesn't just make a request to your website - that would be inefficient and stupid. Instead, it tries to locate the wp-config.php file and, if found, loads its code into memory. Then it removes the line that requires wp-settings.php and executes wp-config.php using eval(). This step is necessary so that all wp-config.php constants get defined (like db credentials), but the bootstrap process doesn't proceed on its own.

Then it does a few other internal operations, and finally after a moment - it loads wp-settings.php. Almost the entirety of WordPress is loaded in just this file (which you should already know). Some important global variables like $wpdb are instantiated, and plugins are loaded. That's right, every WP-CLI command that loads WordPress also loads all plugins and calls the 'init' hook. This has some important consequences we'll talk about in the "Making Plugins WP-CLI Compatible" section.

That's where similarities with the normal request lifecycle end. You can see that the only steps shared are step 4 and 5. The code parsing the request doesn't run, so the global WP_Query isn't instantiated. Similarly, the template loader (responsible for finding the template to render) doesn't run either, which shouldn't really be a surprise. The command executes in this WordPress environment (run after wp-settings.php) and the process finishes.

Note that all of this happens on the system that the website is running on. You can't just run a WP-CLI command on any remote WordPress website. This means that you must have WP-CLI installed on your server for it to work. There is a way to use it over SSH, but it still needs to be installed on the server.

Custom Commands

The built-in commands are enough to do everything you can do in the native wp-admin. But the wp-admin isn't only the native stuff. It's heavily extended by pretty much all plugins. What if you made a backup plugin that allowed users to take full backups of their site with a click of a button? That's an action in wp-admin. If your plugin is big enough (i.e., professional), you might think about creating a custom WP-CLI command for that.

I'm only going to scratch the surface of creating commands. Go read the docs when actually doing it. A command is basically just a name (the command you write after "wp") mapped to a callback, like a function, class, or closure. Commands can be bundled with plugins or they can be external packages. These packages are installed and managed on the system with "wp package". Packages to WP-CLI are like plugins to WordPress.

Probably the most robust method for callback definition is with a class. That's because every public method of the class automatically becomes a subcommand. This is useful if you need many subcommands. Command definitions are pretty verbose because PHPDoc is used to create help for the command. Here is an example of a simple command:

```
<?php
if ( defined( 'WP_CLI' ) && WP_CLI ) {
    class My_Command {
        /**
         * Prints a greeting.
         *
         * ## OPTIONS
         *
         * <name>
         * : The name to greet.
         *
         * [--yell]
         * : If set, the greeting will be uppercase.
         *
         * ## EXAMPLES
         *
         *     wp mycmd greet world --yell
         *
         * @when after_wp_load
         */
        public function greet( $args, $assoc_args ) {
            list( $name ) = $args;
            $greeting = "Hello, $name";
```

```
            if ( isset( $assoc_args['yell'] ) ) {
                $greeting = strtoupper( $greeting );
            }

            WP_CLI::success( $greeting );
        }
    }

    WP_CLI::add_command( 'mycmd', 'My_Command' );
}
```

This command is defined in WordPress (e.g., in a plugin). You can tell because of the check for the WP_CLI constant. This constant is only true if WordPress is loaded in the context of WP-CLI. You don't have to hook that code to any action. It can just run when your plugin is loaded in wp-settings.php. An example usage could be:

```
wp mycmd greet Wiktor --yell
> HELLO, WIKTOR
```

You can see that the greet() method was automatically registered as a subcommand to "mycmd". Also, notice the "@when after_wp_load" at the bottom of the PHPDoc comment. It tells WP-CLI when in the bootstrap process to execute the callback. "after_wp_load" is actually the name of a hook (not a WordPress hook - WP-CLI hook). WP-CLI uses a very similar hook system to WordPress.

"after_wp_load" is the default hook, so it's redundant to specify it. The most important hook you can use here is "before_wp_load". This will execute the command before the WordPress bootstrap process. That's of course only in theory. Keep in mind that this command is added in a WordPress plugin. This means that for WP-CLI to even be aware of it, the WordPress must've already been loaded. In reality, execution before WP loads can only be achieved if you make your command a package (as opposed to bundling it with a plugin).

Making Plugins WP-CLI Compatible

As is the case with many other things, the most important part of making your plugin WP-CLI compatible is knowing you should care about it in the first place. Sometimes your plugins should behave differently when WordPress is loaded in the context of WP-CLI. One such example could be a plugin that counts the number of visitors. You should know how your plugin's logic should change when WordPress is loaded by WP-CLI. Check for the "WP_CLI" constant to do something conditionally (or run your code on actions that WP-CLI doesn't reach).

The biggest difference between a normal bootstrap and WP-CLI bootstrap is the fact that the context is not an HTTP request. This means that HTTP-specific superglobals, like $_SERVER, $_GET, $_POST, etc., might not be populated with values you'd expect in a normal request. If your plugin has any code that runs during WP-CLI (e.g., on init) that blindly uses these values without checking if they exist, you might encounter warnings or errors. The same goes for other HTTP-specific code (like sending headers).

Last but not least, don't echo, print, or die when WP-CLI is running. Printing text directly may interfere with the command output and make it unreadable (i.e., when --format=json is used). If you have to print something, you can use WP_CLI::log() or WP_CLI::error() when you detect the WP-CLI context.

And don't forget to test your plugin with WP-CLI!

Useful Packages

This section is very pragmatic and specific, but I want to give you a broad idea of what's possible with external WP-CLI packages by showing you 3 life-changing ones.

wp doctor helps you diagnose problems on WordPress installations. It runs a series of configurable checks. For example, you can check if:

- the number of cron jobs is within normal range,
- the size of autoloaded options doesn't exceed a specified threshold,
- the WP_DEBUG constant is set to false,
- core needs an update,
- the number of plugins is too large,
- the core WordPress files have been tampered with,
- and much more…

And these are only the built-in checks. The doctor package allows you to write custom doctor.yml configuration files with checks stitched together from a predefined list of options, or you can write completely custom checks with PHP. For example, you can check if a given option in the database has the value you expect, or if a particular plugin is active. Best part? You can specify --format=json and pipe it to a custom script. Run a cron job like that once a day and notify yourself when asserts fail.

wp profile lets you profile WordPress execution. By "profiling" I mean measuring things like execution time (probably the most important feature). You can run "wp profile stage" to see how long it takes for each stage of WordPress to execute. You can also run "wp profile hook" to see how long it takes for each individual hook to execute and which callbacks are taking what amount of time. If you've ever had to deal with a slow WordPress website and you just learned about it, your jaw should be on the floor (mine sure was). Just look at that (--spotlight filters out zeroish values):

```
wiktor@Komputer:/var/www/html/generatepress$ wp profile stage
+------------+---------+------------+-------------+-------------+------------+--------------+-----------+------------+--------------+---------------+
| stage      | time    | query_time | query_count | cache_ratio | cache_hits | cache_misses | hook_time | hook_count | request_time | request_count |
+------------+---------+------------+-------------+-------------+------------+--------------+-----------+------------+--------------+---------------+
| bootstrap  | 0.8453s | 0.0154s    | 56          | 97.29%      | 3739       | 104          | 0.5168s   | 109208     | 0s           | 0             |
| main_query | 0.0043s | 0.0021s    | 9           | 70.27%      | 78         | 33           | 0.0016s   | 263        | 0s           | 0             |
| template   | 0.0964s | 0.0079s    | 27          | 95.6%       | 1129       | 52           | 0.0631s   | 11265      | 0s           | 0             |
+------------+---------+------------+-------------+-------------+------------+--------------+-----------+------------+--------------+---------------+
| total (3)  | 0.946s  | 0.0254s    | 92          | 87.72%      | 4946       | 189          | 0.5815s   | 120736     | 0s           | 0             |
+------------+---------+------------+-------------+-------------+------------+--------------+-----------+------------+--------------+---------------+
wiktor@Komputer:/var/www/html/generatepress$ wp profile hook --spotlight --orderby=time
+-------------------------------+----------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| hook                          | callback_count | time    | query_time | query_count | cache_ratio | cache_hits | cache_misses | request_time | request_count |
+-------------------------------+----------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| pre_http_request              | 1              | 0s      | 0s         | 0           |             | 0          | 0            | 0s           | 1             |
| wp_get_attachment_image_src   | 1              | 0.0002s | 0s         | 0           | 75%         | 3          | 1            | 0s           | 0             |
| before_woocommerce_init       | 3              | 0.0009s | 0.0002s    | 1           | 75%         | 3          | 1            | 0s           | 0             |
| wp_headers                    | 1              | 0.0009s | 0.0005s    | 3           | 70%         | 7          | 3            | 0s           | 0             |
| woocommerce_register_taxonomy | 1              | 0.001s  | 0.0004s    | 2           | 77.78%      | 7          | 2            | 0s           | 0             |
| upload_mimes                  | 1              | 0.001s  | 0.0004s    | 1           | 50%         | 1          | 1            | 0s           | 0             |
| template_include              | 2              | 0.0019s | 0.0004s    | 2           | 93.94%      | 31         | 2            | 0s           | 0             |
| get_block_templates           | 4              | 0.002s  | 0.0009s    | 1           | 91.67%      | 11         | 1            | 0s           | 0             |
| body_class                    | 2              | 0.0021s | 0.0006s    | 1           | 93.75%      | 15         | 1            | 0s           | 0             |
| widgets_init                  | 4              | 0.0102s | 0s         | 0           | 100%        | 106        | 0            | 0s           | 0             |
| wp_loaded                     | 21             | 0.0118s | 0.004s     | 3           | 85.37%      | 35         | 6            | 0.0049s      | 1             |
| wp_enqueue_scripts            | 19             | 0.0134s | 0s         | 0           | 97.78%      | 88         | 2            | 0s           | 0             |
| after_setup_theme             | 12             | 0.0134s | 0.0004s    | 1           | 95.45%      | 63         | 3            | 0s           | 0             |
| wp_print_footer_scripts       | 4              | 0.0177s | 0.0014s    | 5           | 95.53%      | 171        | 8            | 0s           | 0             |
| wp_footer                     | 14             | 0.0188s | 0.0014s    | 5           | 95.94%      | 189        | 8            | 0s           | 0             |
| wp_head                       | 33             | 0.0199s | 0s         | 0           | 98.69%      | 151        | 2            | 0s           | 0             |
| plugins_loaded                | 18             | 0.0386s | 0.0013s    | 5           | 88.89%      | 56         | 7            | 0s           | 0             |
| init                          | 156            | 0.165s  | 0.0045s    | 19          | 95.88%      | 1071       | 46           | 0s           | 0             |
| rest_api_init                 | 23             | 0.3816s | 0.005s     | 22          | 97.35%      | 808        | 22           | 0s           | 0             |
+-------------------------------+----------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| total (19)                    | 320            | 0.7002s | 0.0212s    | 71          | 87.67%      | 2816       | 116          | 0.0049s      | 2             |
+-------------------------------+----------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
wiktor@Komputer:/var/www/html/generatepress$ wp profile hook body_class --spotlight --orderby=time
+----------------+-----------------------------------------------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| callback       | location                                            | time    | query_time | query_count | cache_ratio | cache_hits | cache_misses | request_time | request_count |
+----------------+-----------------------------------------------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| wc_body_class() | woocommerce/includes/wc-template-functions.php:317 | 0.0014s | 0.0002s    | 1           | 93.33%      | 14         | 1            | 0s           | 0             |
+----------------+-----------------------------------------------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
| total (1)      |                                                     | 0.0014s | 0.0002s    | 1           | 93.33%      | 14         | 1            | 0s           | 0             |
+----------------+-----------------------------------------------------+---------+------------+-------------+-------------+------------+--------------+--------------+---------------+
```

wp faker helps create dummy data for testing purposes. It automatically generates things like posts, pages, authors, attachments, categories, and tags, but also WooCommerce products, brands, reviews, and more. You can use it to get realistic content quickly. It's useful in test/dev environments, where you can see how your theme looks on actual data without having to manually input it.

Closing Words

If you just read the entire guide - you should be proud of yourself. Your awareness of different WordPress features is now better than that of most professional WordPress developers. You should not only have a good idea of how to fulfill standard requirements, but also have a deep understanding of how different parts of this system work together. You may not be an expert WordPress developer yet, but I think you can see how this knowledge will help you get there 10x faster.

I really hope you had as much fun reading this guide as I had writing it. I had never created anything of this scale before, and now having done that I can tell you - it's a really stimulating and satisfying experience. Moreover, I have learned an insane amount of things about WordPress by having to go through all the documentation and some of the source code.

My main goal with this guide was always to teach myself, but over time, I started prioritizing the reader (you) more and more. My only hope is that you feel like reading this document was not a waste of your time. Whether that's the case or not, feel free to contact me at

wiktor@wiktorjarosz.com or any other way. You can tell me you love my writing or you can tell me I should go back to primary school - I'll try to answer with something nice either way.

Cheers,
Wiktor

My GitHub: wiktorjarosz
My LinkedIn: Wiktor Jarosz
My blog: wiktorjarosz.com